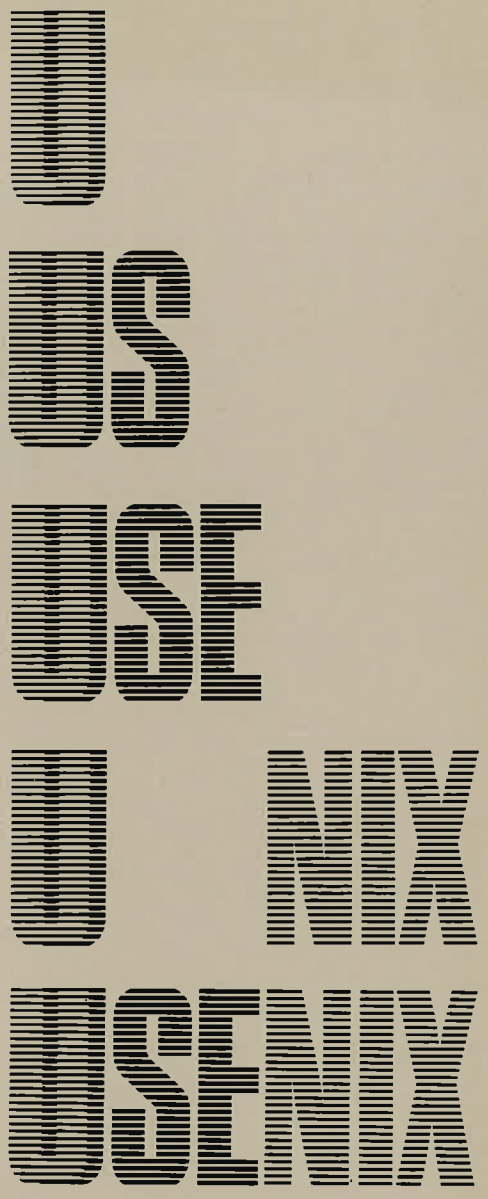
USENIX

MOBILE & LOCATION-INDEPENDENT COMPUTING SYMPOSIUM

Summer 1993



SYMPOSIUM PROCEEDINGS

Mobile & Location-Independent Computing

August 2-3, 1993
Cambridge, Massachusetts

Table of Contents

Mobile & Location-Independent Computing Symposium
Cambridge, Massachusetts
August 2-3, 1993

Experience

Program Committee

Disconnected Operation for AFS

L.B. Huston
lhuston@citi.umich.edu

P. Honeyman
honey@citi.umich.edu

Center for Information Technology Integration
The University of Michigan
Ann Arbor

ABSTRACT

AFS plays a prominent role in our plans for a mobile workstation. The AFS client manages a cache of the most recently used files and directories. But even when the cache is hot, access to cached data frequently involves some communication with one or more file servers to maintain consistency guarantees. Without network access, cached data is soon rendered unavailable.

We have modified the AFS cache manager to offer optimistic consistency guarantees when it can not communicate with a file server. When the client reëstablishes a connection with the file server, it tries to propagate all file modifications to the server. If conflicts are detected, the replay agent notifies the user that manual resolution is needed.

Our system brings the benefits of contemporary distributed computing environments to mobile laptops, offering a fresh look at the potential for nomadic computing.

1. Introduction

The continuing miniaturization of computer hardware hath wrought dramatic changes in portable computers. In tandem, the ties to wall jacks and other wiring requirements are being unbound, which has led to an explosion of interest and activity in nomadic computing. Yet, operating system support for mobile computers has not kept pace, as the research and commercial computing communities have embraced the distributed computing paradigm, in which network connectivity forms a fundamental technological underpinning. To close the gap between these advances in hardware and software, the Center for Information Technology Integration (CITI) has sponsored the LITTLE WORK project [1], which is investigating the operating system requirements for nomadic computers.

The goal of the LITTLE WORK project is to build a mobile computing platform that closely resembles CITI's desktop computing environment. Our current prototype is an Intel i386-based laptop computer running Mach 2.5 from Carnegie Mellon University [2], along with MIT's X Window System [3]. When traveling, we use modems and SLIP [4] to run conventional network-based services such as TCP/UDP/IP, NTP [5], Kerberos [6], and AFS [7].

A key component of our mobile workstation is the distributed file system. There are numerous benefits to using a caching distributed file system instead of a standalone file system on a mobile machine. In the latter case, a user preparing for a trip with her laptop must select the files she will need and manually copy them to the laptop. On her return, she must copy any modified files back to permanent storage, usually a file server or the local disk on her desktop machine.

Compare this to a caching distributed file system. To prepare for a trip, the user attaches the laptop to the network and runs the applications that she intends to use while traveling. This causes the caching mechanism to copy the latest version of the referenced data to the local disk if it is not already there. After heating up the cache, she disconnects from the network and hits the road. Upon arriving at a location that supports

network connectivity, possibly her home base, she establishes a network connection and directs the file system to propagate her modifications to the file server.

The first benefit of the distributed file system approach is that it reduces the potential for human error. The cache manager logs all file modifications, so the user need not worry about forgetting to copy files to and from permanent storage. Furthermore, the distributed file system offers the potential to detect conflicts that arise if shared files are modified by more than one party, resolves those conflicts that it can, and reports all others.

Another advantage is that a distributed file system adapts as a user's work habits change. Laptops tend to be constrained in disk space, limiting the number of applications that can be installed. In a traditional system, when the user wants to use a different application, she must manually install a new program. As part of this installation process she may need to make space for the new application by removing some other files. She may later regret her selection. In a distributed file system, applications are installed by system administrators and are accessed from any machine. The cache manager takes care of copying the necessary files to the local disk, as well as removing files that haven't been used for a long time.

For our distributed file system we use AFS from Transarc. While AFS works well in a desktop environment, it fails completely in a partitioned network. The difficulty is that AFS consistency guarantees require the client cache manager to maintain network connectivity with the servers responsible for data in the client cache. When a server gives some data to a client, the server also issues a "callback." This callback is a promise that the server will notify the client if the cached file is modified. The client uses possession of this callback to ensure that the cached version of the data is the most recent. If a network partition disrupts communication between the client and the file server, the cache manager can not be sure that the server has not tried to revoke any callbacks. The cache manager assumes the worst case: all cached data becomes invalid. At this point, the cache manager refuses to allow access to any of its cached data, even though the preponderance of the data is valid.

Mobile computers enjoy only sporadic network connectivity, and require AFS to be more resilient to network partitioning. The cache manager must allow access to cached copies of files and directories, performing the necessary consistency checks only when a network connection is established with the file server.

2. Related Work

Previous work on disconnected access to a distributed file system includes the Coda project at Carnegie Mellon University [8,9,10]. Coda is a distributed file system similar to AFS, with additional support for server replication. Voluntary disconnection by a client is treated as a special case of network partitioning. A disconnected client can continue working by using any data is has in its cache. When the client reconnects to a network, it gives the collection of updates made while disconnected to an agent that reintegrates the updates into the file system using methods similar to those used to resolve updates across replicated servers. Because Coda enlists support from the file server in reintegrating client updates, it can offer strict transactional guarantees on the entire collection of updates.

UCLA's Ficus replicated file system [11] also supports a form of disconnected access. Ficus uses peer-to-peer operations on replicated files instead of second-class replication (*i.e.*, caching) by clients of a centralized file server. As a replicated server, a mobile Ficus workstation can achieve many of the goals of disconnected operation.

Although Coda and Ficus provide for service while disconnected, we have elected to go our own way with AFS, for several reasons. Most importantly, AFS is where our files are; it would not make sense to use a file system that we don't use every day. We have been satisfied AFS users for many years, and we understand its behavior, so our choice is to adapt AFS for disconnected operation. A related reason to stay with AFS is that there are currently over 75 different cells[1] accessible from our workstations. If we switched to another file system, we would lose the ability to use these cells while disconnected. In short, were we to go with another file system, we would lose access to our own and CITI's resources and the home directories of our colleagues.

[1] A *cell* is an AFS administrative boundary, comparable to a Kerberos *realm*. Some examples of cells are `umich.edu`, `citi.umich.edu`, `alw.nih.gov`, and `cern.ch`.

3. Design

Before modifying AFS, we developed some basic design criteria. First, we prohibit any changes to the AFS servers, restricting our effort to the client cache manager. If our modifications included any changes to the file servers, we would lose the ability to access other AFS cells. The decision to make the client solely responsible for disconnected operation is significantly different from the approach taken by Ficus and Coda. We feel that the benefit of continued access to AFS outweighs potential problems.

Our second design rule is that we are concerned only with disconnected operation. Coda and Ficus provide a high degree of availability by replicating servers, as well as allowing systems to use local data when a file server is not reachable. But disconnected clients exist in a network partition containing a single machine; replicated servers do not provide an advantage for nomadic computing.

This leaves as the primary issue how to modify the cache manager. AFS tries to provide strict consistency guarantees, to wit, AFS guarantees that a client opening a file sees the data stored when the the most recent writer closed the file. This guarantee is hard to honor in a partitioned network. The main difficulty arises because AFS is pessimistic. In the absence of firm evidence to the contrary, the cache manager assumes that cached data is invalid. Our approach is to modify the cache manager to be more optimistic, allowing access to cached files. This needs to be done with care. While it is acceptable to create an inconsistent view of the file system in the cache of the mobile machine, we don't want to store any data back to the file server that will violate the AFS consistency guarantees.

The archetypal conflict occurs when an object modified by a connected client is also modified by a disconnected client. If the disconnected client blindly stores data back to the file server, then the other client's modifications will be overwritten. In practice these types of conflicts are rare; for example, Ousterhout *et al.*, showed that under work loads similar to ours, write sharing rarely occurs [12], and later studies agree that in such environments, write sharing remains rare [9, 13]. Our optimism has also been validated by running simulations using traces gathered from file servers that we access on a daily basis [14].

4. Running Disconnected

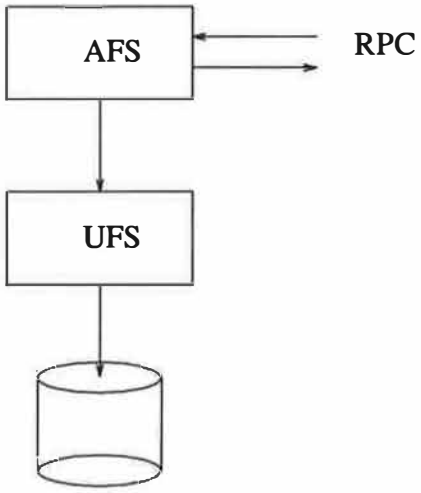
To make disconnected AFS work, we made several changes to the cache manager. One change is to make the cache manager optimistic about cache consistency. We elected to make these modifications at the vnode layer [15]. There are several reasons for this decision. First, it is easier to determine the "correct" behavior because we have access to all the cache manager's information.

In contrast, consider the option of masquerading as a file server at the remote procedure call (RPC) layer. The response to an RPC depends on the the current contents of the cache, as well as other data structures maintained by the cache manager. For example, suppose a user is trying to access a file for which the cache manager no longer has a callback. Normally the cache manager would issue a `getstatus` RPC to the file server, which returns a callback and the current version number of the file. While disconnected, it is not possible to issue this RPC, so we would like the cache manager to assume (optimistically) that its cached data are valid. From the RPC layer, this is difficult because the response must be based on the current contents of the cache. If the cache manager doesn't have a cached copy of the file's status, then the request must fail. But if the file's status is cached, then the `getstatus` request should return the cached information. In either case the RPC layer needs to have knowledge that is easily accessible at the vnode layer.

Another reason to make the modifications at the vnode layer is to aid in propagating file modifications back to the file server. Operations to AFS files consist of a pair of vnode operations: local ones, or `ufs_vnodeops`, and remote ones, or `afs_vnodeops`. When an operation is applied to an AFS file, the appropriate vnode operations are called. The `afs_vnodeop` checks cache validity, fetches current versions of the files, stores any modifications back to the file server, *etc*. Communication with file servers is through the RPC layer.

After the AFS portion of the vnode operation is completed, `ufs_vnodeops` are called on to manipulate the cached data. Consider a `read` operation: the AFS vnode operation ensures that the cached copy of the data is valid, whereupon the UFS vnode operation, `ufs_read`, is called on to return the data out of the local cache.

When the cache manager is disconnected, `ufs_vnodeops` are performed while `afs_vnodeops` are logged and deferred. When a network connection is established, the cache manager iterates through the log



AFS vnode architecture. The AFS vnode operations use the RPC layer to communicate with the file server, then call the underlying UFS vnode operations to access the data in the local disk cache.

of deferred `afs_vnodeops` and tries to replay the operations to the file server. By logging at the vnode layer, we are assured that after replaying the log, all of the `ufs_vnodeops` and `afs_vnodeops` have been performed. In the absence of any conflicts, we will see the same final state that would have resulted had the operations been performed while connected.

This creates a good basis for the replay algorithm, but we need to modify the algorithm to account for operations that might violate the AFS consistency guarantees. This issue is discussed in the next section.

To put the cache manager into disconnected mode, the user issues a `disconnect` command. Thereafter, every successful vnode operation on a file generates a corresponding log entry; operations that fail are not logged. Along with the type of operation, enough extra information is logged to allow the replay agent to execute the same operation when reconnecting with the file server. The extra information depends on the type of operation, but is typically such data structures as file name, AFS' internal file identifier, and current data version number. If the operation is one that modifies the state of the file system, the files in the local cache are modified to reflect the change.

The principal difference in the cache manager's behavior when running disconnected is the way it enforces consistency. Ordinarily, before the cache manager references an object, it assures that it has a callback for that object; if not, then the cache manager issues a `getstatus` RPC to the server to get a callback. When the cache manager is disconnected, it can not perform this or any other RPC. Instead it marks the object as having a callback issued by the local host. (Normally a callback is marked with the IP address of the server responsible for the object.) The locally issued callback is used until AFS is reconnected.

In disconnected mode, the occasional cache miss is an inevitable fact of life. Under normal circumstances, the cache manager asks a file server for the missing information. As a disconnected cache manager can not get this information, it returns an appropriate error (`ENETDOWN`) to the calling program.

In our travels with a LITTLE WORK along, we occasionally find that we are missing an important file, yet we are not willing to pay the price of replaying a substantial log, so we have added a "fetch-only" mode of disconnected operation. In fetch-only mode, the cache manager issues `afs_vnodeop` RPCs for non-mutating operations, and logs mutating `afs_vnodeops`.

Although early versions of AFS cached whole files, the current version breaks large files into 64 KB chunks. This chunking lets an AFS client work with files larger than the local disk. This has scant advantage for a mobile computer, and poses potential problems should we discover that an essential file, say `/usr/X11/bin/X386`, is only partially resident in the cache. Our inclination is to revert to whole-file caching; as an expedient (read *hack*), we set the chunk size on our mobile clients to be one megabyte.

Another problem we have encountered is applications that unnecessarily demand more information than the disconnected cache manager can provide. A potent example is the UNIX file removal program, `rm`. In

this code fragment, taken from the Berkeley UNIX `rm.c`, the program obtains status information about the target to be removed, so that it can complain about a request to remove a directory or a protected file.

```
if (lstat(arg, &buf)) {
   if (!fflg) {
      fprintf(stderr, ...);
      errcode++;
   }
   return (0);    /* error */
}
```

If we try to remove a file for which we don't have status information cached, `rm` fails. (Observe that the `-f` flag does nothing more than make `rm` fail silently, in this case.) Yet the underlying vnode operations are capable of performing the operations required to remove the file, although there are complications on replay. We have modified our copy of `rm.c`, but we're not happy about it.

5. Replaying the Log

After running disconnected for a while, a network connection must be established to propagate modifications back to the file server. Once a connection is established, the user issues the `reconnect` command. This command iterates through the log of deferred `afs_vnodeops` and attempts to perform all the delayed operations.

A problem arises when the same file is modified by a connected client and a disconnected client. Before replaying an operation that changes the state of the file server, say a `storedata` RPC, the replay agent checks to see whether the data has been modified by another client. If so, we consider the case to be an instance of concurrent write sharing.[2] We don't have tools to resolve such a conflict, so the replay agent is left with the responsibility for preserving both the modified data on the file server and the modified data in its cache. Our solution is to store the locally modified data on the file server in a renamed object, so that both versions are stored on the file server. The replay agent also lets the user know a conflict has occurred, and encourages her to resolve the problem by hand.

AFS vnode operations can be divided into two classes. The first class is the non-mutating operations: these operations do not cause any changes to file server state. An example of this type of operation is `afs_read`. Non-mutating operations can not be involved in write conflicts, and don't require information transfer to the file server at replay. Nonetheless, we log these operations because some useful information can be determined from them, *e.g.*, we can detect when stale data may have been used.

The second class of operations are the mutating ones, those that modify the file server's state. An example of a mutating operation is `afs_create`. Mutating operations pose the most concern, as they have the potential to be involved in write conflicts.

In the next sections, vnode operations of both classes are listed along with the appropriate methods for performing the replay and detecting conflict. Resolving data conflicts is left to the user, but in many cases, the replay manager can resolve directory conflicts on its own.

5.1. Non-mutating operations

Non-mutating operations are logged to help determine if stale data was used. Non-mutating operations do not cause any changes on the file servers, and can not be involved in write conflicts. However, it is possible for non-mutating operations to be performed on stale data. The goal in replaying non-mutating operations is to detect any stale data usage so that the user can be warned appropriately.

Some of the algorithms below use timestamping to compare the modification times of files with the time that the operation was performed. This can be troublesome because a disconnected client has no means of running a time synchronization protocol. Consequently we need to rely on the laptop clock not drifting too far. To help account for drift we consider changes within a time window instead of a single instance in

[2] We don't assume strictly synchronized clocks, so we can not use file modification timestamps to discriminate sequential write conflicts from concurrent ones. Otherwise, we might consider the traditional UNIX approach: "last writer wins."

time.

5.1.1. afs_open

To determine if a file was stale when it was opened, the replay agent fetches the current version number of the file from the server and compares it to the version number of the file when it was cached. If the version numbers are identical, the file was valid when used locally. If they differ, stale data may have been used; this depends on whether the `open` preceded the modification of the file.

Because an AFS server increments file version numbers by one each time a file is modified, the replay agent can tell how many times the file was modified on the server. If the file was modified more than once, it is impossible to know the earliest modification time, as only the latest modification time is stored. If the file was modified exactly once, a comparison of the modification time with the local open time determines whether stale data was used. If the file was modified more than once, and the local open time does not precede the modification time, it is possible that stale data was used, but we can't know.

Using these rules, the replay agent reports to the user whether stale data was used or may have been used.

5.1.2. afs_read

The rules for determining the use of stale data are applied at the time the file is opened, not when it is read, so no warnings are issued here.

5.1.3. afs_lookup

It is possible to use the same algorithm used by `afs_open` to determine whether stale data was used during a lookup. However, this information may not provide any useful information to users, and because of their high frequency, logging potential lookup conflicts might generate so much output that it would obscure important messages from the replay agent. Therefore, we do not report stale lookups, but we do record them for our own analysis.

5.1.4. afs_getattr

Attribute modification times are not exported via AFS, so the replay agent can not know when the attributes were changed. As a heuristic to determine whether `afs_getattr` used stale data, the replay agent compares the current attributes with the attributes of the cached file. If the attributes have changed since the last time they were cached, the replay agent informs the user that stale attributes may have been used.

5.1.5. afs_readlink, afs_access, afs_readdir

The issues for these vnode operations are similar to `afs_lookup`. It is possible to determine if stale data was used, but this does not seem to provide any useful information for the user. Therefore, we treat these operations as we do `afs_lookup`.

5.2. Mutating operations

The mutating vnode operations offer the most challenge. These operations modify server data, so they have the potential to be involved in write conflicts and violations of AFS guarantees.

Most conflicts are resolved by creating a new instance of the object, and dumping the contents of the disconnected version into the newly created object. To construct a new name for an object, the replay agent modifies the original name by repeatedly appending a suffix until the new name is unique in its directory. To provide the user with a hint about the origin of the new file, the suffix reflects the type of operation being performed. The replay agent keeps track of such renaming, so that later operations on the same object are directed to the right file name. For example, if the replay agent is forced to store changes to `foo` in the new file `foo.ren`, later attribute changes to `foo` are applied to `foo.ren`.

Sometimes it is not possible to replay the operations as they were performed while disconnected. For example, if a file and its parent directory are removed from the file server, modifications to that file by a disconnected client will have no place to be stored. Yet preservation of data is a key goal of our replay algorithm. To handle these problems we provide a centralized location called the "orphanage" to store such files.

5.2.1. afs_create

If the parent directory no longer exists, then the replay agent creates the file in the orphanage. If the parent exists and the file already exists, then the replay agent iteratively appends ".creat" to the name enough times to assure uniqueness, and creates a file with the name that results. If there is no conflict, then the cache manager creates a file in the normal manner.

5.2.2. afs_write

Although afs_write operations modify the client cache, writes are not propagated to the file server until the file is closed. This means that we can ignore afs_writes during replay and be assured that the right thing will happen when the file is closed.

5.2.3. afs_close

We can safely ignore any afs_close operations performed on files opened only for reading. If the file was open for writing, modifications to the file must be propagated back to the file server. We compare the version number of the cached file to the server's version number for the file. If they are the same, then the file has not changed since we cached it, so we store our changes.

If the version number of the file has changed, then we do not overwrite the copy on the file server, not even if we are convinced that the local user is the "last writer." Instead, the replay agent creates a new file with a unique name and copies the data from the cache into this new file and notifies the user.

If the parent directory was removed while we were disconnected, a new version of the file is created in the orphanage, and the cached data is copied into this new file.

5.2.4. afs_mkdir

The afs_mkdir call is similar to afs_create: the replay agent must ensure that no entry in the parent directory has the same name as the directory being created. If there is no conflict, then the afs_mkdir can proceed normally.

If there is a conflict, then a unique name is generated and a directory is created with this new name. If the parent directory no longer exists, then the new directory is created in the orphanage.

5.2.5. afs_remove

To replay afs_remove, the replay agent compares its version number of the file being removed with that on the server. If they are the same, the file is removed. If they differ, the file has been modified. We must not remove it as this would destroy someone else's fresh data.

To allow files to be removed without having corresponding attribute information in the cache, we invent an invalid version number as the cached version number. In this case, remove fails on replay, forcing the user to reissue the rm command.

5.2.6. afs_rmdir

Because only successful operations are logged, the locally cached copy of the target directory must have been empty at the time the rmdir operation was issued. Therefore, the replay agent can simply issue the afs_rmdir operation, assured that the operation will succeed or fail on the server appropriately. If the operation fails, the failure status is reported to the user.

5.2.7. afs_link

Like afs_create and afs_mkdir, if the target does not yet exist, then afs_link proceeds normally. Otherwise, we generate a unique name and create the link with this new name.

5.2.8. afs_symlink

This is the same as afs_link, except a symbolic link is being created instead of a hard link. The replay agent needs to ensure that the name is unique, then create the symbolic link.

5.2.9. afs_fsync

This operation forces modified data to be stored on the file server. The replay agent must perform the same consistency checks as for afs_close.

5.2.10. afs_setattr

In replaying afs_setattr, the replay agent needs to make sure that no other changes to the attributes are lost. It needs to compare the current attribute information with the cached attribute information. If there are no differences the replay agent assumes there is no conflict and proceeds with the setattr.

If there are differences between the current version and the cached version, the setattr is aborted and the user is informed of the conflict. It would be possible to merge the two versions of the new attributes if they do not conflict, but this might lead to some unpredictable side effects. Instead, we abort the attempt and alert the user.

5.2.11. afs_rename

The afs_rename operation deletes the target if it already exists; the cache manager logs whether this was the case when the afs_rename operation was initially issued. If the target did not exist, and if a fresh target does not exist on the server when the replay agent runs, then the afs_rename operation is issued to the server. If the target does exist on the server, then its name is modified to make it unique, and the afs_rename operation is run with the new target name.

As with afs_remove, the disconnected cache manager allows a rename if a destination entry exists but no information for this entry is cached. In this case, the destination version number is marked as invalid so replay will fail to destroy this file.

6. Data Persistence

One problem encountered in our AFS modifications is the amount of state that the cache manager builds while talking to the servers. The cache manager stores each object in two components: the vcache, which holds status and callback information associated with the object; and the dcache, which holds the information needed to translate AFS' internal name for a file into a name in the local cache.[3]

AFS needs a dcache for every file in the cache. There are too many to hold in kernel memory, so the cache manager stores most dcaches on disk and moves them into a memory cache as needed. vcaches are not as critical, so the cache manager keeps a large cache of them in kernel memory. AFS was designed to run on high-speed networks, so the expense of fetching vcaches from the file server has traditionally been negligible. But in a disconnected environment, it is not possible for the cache manager to ask the file server for the current vcaches, so we store vcaches on disk just as we do dcaches, and move them into memory when they are referenced. This also allows vcache information to survive reboots.

In addition to the vcache and dcache entries, the cache manager keeps other important information in volatile memory. The AFS namespace is constructed from subtree components called *volumes* [16], which are mounted together to form AFS' hierarchical name space. To cross a mount point, the client must obtain a mapping from a volume name to a directory identifier and a server name. Ordinarily, this mapping is obtained from a volume location database server. Once a mount point has been crossed, the client caches the mapping to speed subsequent lookups. However, this information is kept in volatile memory, so it does not survive a reboot.

The cache manager needs these volume mappings and other memory based data structures to access cached files. For mobile clients to continue working across reboots, we had to make these data structures persistent, again using disk files as backing stores for these data structures. These data structures are loaded into memory from the disk files as part of the AFS start up procedure.

[3] This internal name, called a FID, is a data structure consisting of a cell identifier, a volume name, a vnode number, and a version number, *e.g.*, <8DD3A818, 200024A1, 1C8A, 12>. The local file name looks like /usr/vice/cache/V1001.

7. Current Status

We are running disconnected AFS on our laptop machines and use it on a daily basis. We have most of the bugs worked out and are experimenting with disconnected operation to help determine what features need to be added. Disconnected AFS was used in writing portions of this paper.

8. FUTURE WORK

There are several areas that we plan to address in the near term. One issue is callback management. We would like to add support to the cache manager so that when it is connected it tries to keep the most up-to-date version of the files in our cache. This will help reduce potential replay conflicts, because we won't start by using stale data.

Another issue that needs to be addressed is the cache replacement policy. AFS currently uses a least-recently-used (LRU) policy to determine which items in the cache should be replaced. We will use our nomadic experiences to determine if this works well enough for our needs. We may need to be able to "pin" certain files in the cache, *e.g.*, system binaries used to set up a network.

We also plan to provide tools to help users resolve conflicts. It is sometimes possible to determine an effective heuristic behavior for a specific application. For example, we make heavy use of the `MH` mailer, which stores each mail message in a sequentially numbered file. If messages are inserted simultaneously by a connected client and a disconnected client, we might end up with a message file called `1234.creat`. Yet we know how to fix matters: rename the file to an unused number. We plan to write scripts to address this and other application-specific scenarios that we encounter.

Another area that needs work is management of the log file. When running disconnected, many operations are overruled by later operations. For instance, if we create a file while disconnected and remove it before we reconnect, there is no point in storing the file during the replay. We plan to write a program that examines the log and removes this and other extraneous operations. This will have multiple benefits. First, it will reduce the disk space needed by the log. Second, it can reduce the time necessary for replay, and might make replay more palatable over slow or expensive data links.

Another problem we face is how to react to the use of stale data. Ideally, we would like to be able to identify all the files and processes that depended on this stale data, so that the user may be able take some corrective action, such as rerunning the processes on fresh files. To provide this functionality, we would need support for transactions, which could provide us with the information necessary to detect these read-write conflicts.

9. Conclusions

Disconnected operation vastly increases the benefits of nomadic computing. Currently, to use a LITTLE WORK machine, a user needs only to use the laptop on a network for a LITTLE WHILE and the machine is ready to roll. If she performs work similar to what she intends to do on the road, the cache will contain all the files necessary to support her needs. At this point she may tell AFS to run disconnected.

While traveling, the user can use the laptop as if she were sitting at her desk. Files and directories are all in their usual places. When arriving back home or in a hotel room, the user can establish a network connection and tell the cache manager to reconnect, whereupon all changes are propagated back to stable storage. As a result, the nomadic computing environment closely mimics the conventional one, vastly extending the range and scope of mobile computing.

Acknowledgements

We thank Jim Rees, Mike Stolarchuk, Mary Jane Northrup, and M. Satyanarayanan and for their insight and assistance. This work was partially supported by IBM and Telebit.

References

1. P. Honeyman, L. Huston, J. Rees, and D. Bachmann, "The LITTLE WORK Project," *Proceedings of the Third IEEE Workshop on Workstation Operating Systems*, Key Biscayne, FL (April 1992).
2. Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and

Michael Young, "Mach: A New Kernel Foundation for UNIX Development," *USENIX Conference Proceedings*, Atlanta, GA (Summer 1986).

3. R.W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics* **5**(2) (April, 1987).
4. J.L. Romkey, "Nonstandard for transmission of IP datagrams over serial lines: SLIP," RFC 1055, Network Information Center, SRI International, Menlo Park, CA (June 1988).
5. D.L. Mills, "Network Time Protocol (Version 3): Specification, Implementation, and Analysis," RFC 1305, Network Information Center, SRI International, Menlo Park, CA (March 1992).
6. J.G. Steiner, B.C. Neuman, and J.I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," *USENIX Conference Proceedings*, Dallas, Texas (February, 1988).
7. John H. Howard, "An Overview of the Andrew File System," *USENIX Conference Proceedings*, Dallas, TX (Winter 1988).
8. M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers* (April 1990).
9. J.J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Transactions of Computer Systems* **10**(1) (February 1992).
10. P. Kumar and M. Satyanarayanan, "Log-Based Directory Resolution in the Coda File System," *Second International Conference on Parallel and Distributed Information Systems*, San Diego, CA (January 1993).
11. J.S. Heidemann, T.W. Page, R.G. Guy, and G.J. Popek, "Primarily Disconnected Operation: Experiences with Ficus," *Proceedings of the Second Workshop on the Management of Replicated Data* (November 1992).
12. J. Ousterhout, H.L. DaCosta, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A Trace-Driven Analysis of the Unix 4.2 BSD File System," *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (December 1985).
13. Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, "Measurements of a Distributed File System," *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA (October 1991).
14. A.M. Khandker, "Mobile Computing: Running AFS over Dial-up Connections," CITI Tech. Report, University of Michigan (In preparation).
15. S.R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *USENIX Conference Proceedings*, Atlanta, GA (Summer 1986).
16. R.N. Sidebotham, "Volumes: The Andrew File System Data Structuring Primitive," *European Unix User Group Conference Proceedings* (August 1986).

Availability

Researchers with an armful of source licenses may contact `info@citi.umich.edu` to request access to our AFS client modifications.

Experience with Disconnected Operation in a Mobile Computing Environment

M. Satyanarayanan, James J. Kistler, Lily B. Mummert, Maria R. Ebling, Puneet Kumar, Qi Lu
School of Computer Science
Carnegie Mellon University

Abstract

In this paper we present qualitative and quantitative data on file access in a mobile computing environment. This information is based on actual usage experience with the Coda File System over a period of about two years. Our experience confirms the viability and effectiveness of *disconnected operation*. It also exposes certain deficiencies of the current implementation of Coda, and identifies new functionality that would enhance its usefulness for mobile computing. The paper concludes with a description of what we are doing to address these issues.

1. Introduction

Portable computers are commonplace today. In conjunction with high- and low-bandwidth cordless networking technology, such computers will soon provide a pervasive hardware base for *mobile computing*. A key requirement of this new world of computing will be the ability to access critical data regardless of location. Data from shared file systems must be made available to programs running on mobile computers. But mobility poses serious impediments to meeting this requirement.

We begin this paper by describing how shared file access is complicated by the constraints of mobile computing. We then show how the design of the *Coda File System* addresses these constraints. The bulk of the paper focuses on our usage experience with Coda. We present qualitative and quantitative data that shed light on Coda's design choices. Based on our experience, we have identified a number of ways in which Coda could be improved. The paper concludes with a description of our current work along these dimensions.

2. Constraints of Mobile Computing

Access to shared data in a mobile environment is complicated by three fundamental constraints. These constraints are intrinsic to mobility, and are not just artifacts of current technology:

- *Mobile elements are resource-poor relative to static elements.* For a given cost and level of technology, mobile elements are slower and have less memory and disk space than static elements. Weight, power, and size constraints will always conspire to preserve this inequity.
- *Mobile elements are more prone to loss, destruction, and subversion than static elements.* A Wall Street stockbroker is more likely to be mugged on the streets of Manhattan and have his or her laptop stolen than to have the workstation in a locked office be physically subverted. Even if security isn't a problem, portable computers are more vulnerable to loss or damage.
- *Mobile elements must operate under a much broader range of networking conditions.* A desktop workstation can typically rely on LAN or WAN connectivity. A laptop in a hotel room may only have modem or ISDN connectivity. Outdoors, a laptop with a cellular modem may find itself in intermittent contact with its nearest cell.

These constraints violate many of the assumptions upon which today's distributed systems are based. Further, the ubiquity of portable computers will result in mobile computing systems that are much larger than the distributed systems of today. *Scalability* will thus be a continuing concern.

Ideally, mobility should be completely *transparent* to users. Transparency relieves users of the need to be constantly aware of the details of their computing environment, thus allowing them to focus on the real tasks at hand. The adaptation necessary to cope with the changing environment should be initiated by the system rather than by users. Of course, perfect transparency is an unattainable ideal. But that should not deter us from exploring techniques that enable us to come as close as possible to the ideal.

3. Overview of Coda File System

Coda, a descendant of the Andrew File System [4], offers continued access to data in the face of server and network failures. Earlier papers [7, 9, 14, 15, 16, 17] have described various aspects of Coda in depth. Here we only provide enough detail to make the rest of the paper comprehensible.

Coda is designed for an environment consisting of a large collection of untrusted Unix[1] clients and a much smaller number of trusted Unix file servers. The design is optimized for the access and sharing patterns typical of academic and research environments. It is specifically not intended for applications such as online transaction processing applications that exhibit highly concurrent, fine granularity update patterns.

Each Coda client has a local disk and can communicate with the servers over a high bandwidth network. Clients view Coda as a single, location-transparent shared Unix file system. The Coda namespace is mapped to individual file servers at the granularity of subtrees called *volumes*. At each client, a *cache manager (Venus)* dynamically obtains and caches data as well as volume mappings.

Coda uses two distinct, but complementary, mechanisms to achieve high availability. Both mechanisms rely on an *optimistic replica control* strategy. This offers the highest degree of availability, since data can be updated in any network partition. The system ensures detection and confinement of conflicting updates after their occurence, and provides mechanisms to help users recover from such conflicts.

3.1. Server Replication

The first high-availability mechanism, *server replication*, allows volumes to have read-write replicas at more than one server. The set of replication sites for a volume is its *volume storage group (VSG)*. The subset of a VSG that is currently accessible is a client's *accessible VSG (AVSG)*. The performance cost of server replication is kept low by callback-based caching [6] at clients, and through the use of parallel access protocols. Modifications at a Coda client are propagated in parallel to all AVSG sites, and eventually to missing VSG sites.

3.2. Disconnected Operation

Although server replication is an important part of Coda, it is the second high-availability mechanism, *disconnected operation*, that is a key enabling technology for mobile computing [8]. A client becomes disconnected with respect to a volume when no server in its VSG is accessible. An *involuntary disconnection* can occur in a mobile computing environment when there is a temporary impediment to communication. This can be caused by limitations such as short range, inability to operate underground and in steel-framed buildings, or line-of-sight constraints. A *voluntary disconnection* can occur when a user deliberately operates isolated from a network. This may happen because no networking capability is available at the location of a mobile computer, or to avoid use of the network for cost or power consumption reasons.

While disconnected, Venus services file system requests by relying solely on the contents of its cache. Since cache misses cannot be serviced or masked, they appear as failures to application programs and users. The persistence of changes made while disconnected is achieved via an operation log implemented on top of a transactional facility called *RVM* [16]. Venus implements a number of optimizations to reduce the size of the operation log.

To support disconnected operation, Venus operates in one of three states: *hoarding*, *emulation*, and *reintegration*. Venus is normally in the hoarding state, relying on server replication but always on the alert for possible disconnection. The hoarding state is so named because a key responsibility of Venus in this state is to ensure that critical objects are in the cache at the moment of disconnection. Upon disconnection, Venus enters the emulation state and remains there for the duration of disconnection. Upon reconnection, Venus enters the reintegration state, resynchronizes its cache with its AVSG, and then reverts to the hoarding state.

Venus combines implicit and explicit sources of information in its *priority-based cache management*

[1] Unix is a trademark of Unix System Laboratories.

algorithm. The implicit information consists of recent reference history, as in traditional caching algorithms. Explicit information takes the form of a per-client *hoard database (HDB)*, whose entries are pathnames identifying objects of interest to the user at that client. A simple front-end program called `hoard` allows a user to update the HDB directly or via command scripts called *hoard profiles*.

Venus periodically reevaluates which objects merit retention in the cache via a process known as *hoard walking*. Hoard walking is necessary to meet user expectations about the relative importance of objects. When a cache meets these expectations, it is said to be in *equilibrium*.

4. Implementation Status

Disconnected operation in Coda was implemented over a period of two to three years. A version of disconnected operation with minimal functionality was demonstrated in October 1990. A more complete version was functional in early 1991 and began to be used regularly by members of the Coda group. By the end of 1991 almost all of the functionality had been implemented, and the user community had expanded to include several users outside the Coda group. Several of these new users had no connection to systems research whatsoever. Since mid-1992 implementation work has consisted mainly of performance tuning and bug-fixing. The current user community includes about 30 users, of whom about 20 use Coda on a regular basis. During 1992 the code was also made available to several sites outside of Carnegie Mellon University (CMU), and they are now using the system on a limited basis.

There are currently about 25 laptop and about 15 desktop clients in use. The laptops are mostly 386-based IBM PS2/L40's and the desktops are a mix of DECStation 5000/200's, Sun Sparcstations, and IBM RTs. We expect to be adding about 20 newer 486-based laptops in the near future. We currently have three DECstation 5000/200's with 2GB of disk storage in use as production servers, volumes being triply replicated across them. Additional servers are used for debugging and stress-testing pre-release versions of the system.

The production servers currently hold about 150 volumes. Roughly 25% of the volumes are *user* volumes, meaning that they are assigned to specific users who have sole administrative authority over them. Users are free, of course, to extend access rights to others by changing access-control lists on specific objects in the volume. Approximately 65% of the volumes are *project* volumes, for which administrative rights are assigned collectively to the members of a group. Most of the project volumes are used by the Coda project itself, although there are three or four other groups which have some project volumes. The other 10% of the volumes are *system* volumes, which contain program binaries, libraries, header files, and the like.

To limit our logistical and manpower commitments, we use Coda in slightly different ways on our desktop and laptop clients. On desktop clients, Coda is currently used only for user and project data. The system portions of their namespaces are in AFS, and maintenance of these namespaces is by the CMU facilities staff. Disconnected operation on these machines is therefore restricted to cases in which AFS servers are accessible but Coda servers are not. Such cases can arise when Coda servers have crashed or are down for maintenance, or when a network partitioning has separated a client from the Coda servers but not from AFS servers.

Our mobile clients do not use AFS at all and are therefore completely dependent on Coda. The system portions of the name space for this machine type are maintained by us in Coda. To minimize this maintenance effort, we initially supported only a minimal subset of the system software and have grown the size of the supported subset only in response to user requests. This strategy has worked out very well in practice, resulting in a highly usable mobile computing environment. Indeed, there are many more people wishing to use Coda laptops than we can accommodate with hardware or support services.

Porting Coda to a new machine type is relatively straightforward. Most of the code is outside the kernel. The only in-kernel code, a VFS driver [17], is small and entirely machine independent. Porting simply involves recompiling the Coda client and server code, and ensuring that the kernel works on the specific piece of hardware.

5. Qualitative Evaluation

The nature of our testbed environment has meant that we have more experience with voluntary than with involuntary disconnected operation. The most common disconnection scenario has been a user detaching his or her laptop and taking it home to work in the evening or over the weekend. We have also had cases where users have taken their laptops out of town, on business trips and on vacations, and operated disconnected for a week or more.

Although the dependence of our desktop workstations on AFS has limited our experience with involuntary disconnections, it has by no means eliminated it. Particularly during the early stages of development, the Coda servers were quite brittle and subject to fairly frequent crashes. When the crash involved corruption of server meta-data (alas, a common occurence) repairing the problem could take hours or even days. Hence, there were many opportunities for clients to involuntarily operate disconnected from user and project data.

We present our observations of hoarding, server emulation, and reintegration in the next three sections. This is followed by a section with observations that apply to the architecture as a whole.

5.1. Hoarding

In our experience, hoarding has substantially improved the usefulness of disconnected operation. Disconnected cache misses have occurred, of course, and at times they were quite painful, but there is no doubt that both the number and the severity of those misses were dramatically reduced by hoarding. Moreover, this was realized without undue burden on users and without degradation of connected mode performance.

Our experience has confirmed one of the main premises of hoarding: that implicit and explicit sources of reference information are both important for avoiding disconnected cache misses, and that a simple function of hoard and reference priorities can effectively extract and combine the information content of both sources. It also confirms that the cache manager must actively respond to local and remote disequilibrating events if the cache state is to meet user expectations about availability. In the rest of this section we examine specific aspects of hoarding in more detail.

5.1.1. Hoard Profiles

The aggregation of hints into profiles is a natural step. If profiles had not been proposed and support for them had not been built into the `hoard` tool, it's certain that users would have come up with their own ad-hoc profile formulations and support mechanisms. No one, not even the least system-savvy of our users, has had trouble understanding the concept of a profile or making modifications to pre-existing profiles on their own. And, although there has been occasional direct manipulation of the HDB via the `hoard` tool, the vast majority of user/HDB interactions have been via profiles.

Most users employ about 5-10 profiles at any one time. Typically, this includes one profile representing the user's "personal" data: the contents of his or her root directory, notes and mail directories, etc. Several others cover the applications most commonly run by the user: the window system, editors and text formatters, compilers and development tools, and so forth. A third class of profile typically covers data sets: source code collections, publication and correspondence directories, collections of lecture notes, and so on. A user might keep a dozen or more profiles of this type, but only activate a few at a time (i.e., submit only a subset of them to the local Venus). The number of entries in most profiles is about 5-30, with very few exceeding 50. Figure 1 gives examples of typical hoard profiles.

Contrary to our expectations, there has been little direct sharing of profiles. Most of the sharing that has occurred has been indirect; that is, a user making his or her own copy of a profile and then changing it slightly. There appear to be several explanations for this:

- early users of the system were not conscientious about placing application profiles in public areas of the namespace.
- our users are, for the most part, quite sophisticated. They are used to customizing their environments via files such as `.login` and `.Xdefaults` (and, indeed, many cannot resist the temptation to constantly do so).

```
# Personal files
a /coda/usr/satya 100:d+
a /coda/usr/satya/papers/mobile93 1000:d+

# System files
a /usr/bin 100:d+
a /usr/etc 100:d+
a /usr/include 100:d+
a /usr/lib 100:d+
a /usr/local/gnu d+
a /usr/local/rcs d+
a /usr/ucb d+
```

(a)

```
# X11 files
# (from X11 maintainer)
a /usr/X11/bin/X
a /usr/X11/bin/Xvga
a /usr/X11/bin/mwm
a /usr/X11/bin/startx
a /usr/X11/bin/xclock
a /usr/X11/bin/xinit
a /usr/X11/bin/xterm
a /usr/X11/include/X11/bitmaps c+
a /usr/X11/lib/app-defaults d+
a /usr/X11/lib/fonts/misc c+
a /usr/X11/lib/system.mwmrc
```

(b)

These are typical hoard profiles in actual use by some of our users. The 'a' at the beginning of a line indicates an add-entry command. Other commands are delete an entry, clear all entries, and list entries. The numbers following some pathnames specify hoard priorities (default 10). The 'c+' and 'd+' notations indicate meta-expansion, as explained in Section 5.1.3.

Figure 1: Sample Hoard Profiles

- most of our users are working independently or on well-partitioned aspects of a few projects. Hence, there is not much incentive to share hoard profiles.

We expect that the degree of direct profile sharing will increase as our user community grows, and as less sophisticated users begin to use Coda.

5.1.2. Multi-Level Hoard Priorities

The earliest Coda design had only a single level of hoard priority; an object was either "sticky" or it was not. Sticky objects were expected to be in the cache at all times. Although the sticky approach would have been simpler to implement and easier for users to understand, we are certain that it would have been much less pleasant to use and far less effective in avoiding misses than our multi-level priority scheme.

We believe that a sticky scheme would have induced the following, undesirable types of hoarding behavior:

- a tendency to be conservative in specifying hints, to avoid pinning vast amounts of low-utility data.
- a proliferation of hoard profiles for the same task or data set into, for example, "small," "medium," and "large" variants.
- micro-management of the hoard database, to account for the facts that profiles would be smaller and more numerous and that the penalty for poor specification would be higher.

The net effect of all this is that much more time and effort would have been demanded by hoarding in a sticky scheme than is the case now. This would have reduced the ability of users to hoard effectively, resulting in more frequent disconnected misses. Overall, the utility of disconnected operation would have been sharply reduced.

An argument besides simplicity which is sometimes used in favor of the sticky approach is that "you know for sure that a sticky object will be in the cache when you disconnect, whereas with priorities you only have increased probability that a hoarded object will be there." That statement is simply not true. Consider a trivial example in which ten objects have been designated sticky and they occupy 90% of the total cache space. Now suppose that all ten are doubled in size by a user at another workstation. How can the local cache manager ensure that all sticky objects are cached? Clearly it cannot. The best it can do is re-fetch an arbitrary subset of the ten, leaving the rest uncached.

A negative aspect of our current priority scheme is that the range of hoard priorities is too large. Users are unable to classify objects into anywhere near 1000 equivalence classes, as the current system allows. In fact, they are often confused by such a wide range of choice. Examination of many private and a few shared profiles revealed that, while most contained at least two levels of priority, few contained more than three or four. Moreover, it was also apparent that no user employs more than six or seven distinct levels across all profiles. We therefore believe that future versions of the system should offer a priority range of about 1-10 instead of the current 1-1000. Such a change would reduce the uncertainty felt by some users as well as aid in the standardization of priorities across profiles.

5.1.3. Meta-Expansion

To reduce the verbosity of hoard profiles and to simplify their maintenance, Coda supports *meta-expansion* of HDB entries. If the letter 'c' (or 'd') follows a pathname in a hoard profile, the command also applies to immediate children (or all descendants). A '+' following the 'c' or 'd' indicates that the command applies to all future as well as present children or descendents.

Meta-expansion has proven to be an indispensible feature of hoarding. Virtually all hoard profiles use it to some degree, and some use it exclusively. There are also many cases in which a profile would not even have been created had meta-expansion not been available. The effort in identifying the relevant individual names and maintaining the profile over time would simply have been too great. Indeed, it is quite possible that hoarding would never have reached a threshold level of acceptance if meta-expansion had not been an option.

A somewhat unexpected benefit of meta-expansion is that it allows profiles to be constructed incrementally. That is, a usable profile can almost always be had right away by including a single line of the form "`add <rootname> d+`," where `<rootname>` is the directory heading the application or data set of interest. Typically, it is also wise to specify a low priority so that things don't get out of hand if the sub-tree turns out to be very large. Later, as experience with the application or data set increases, the profile can be refined by removing the "root expansion" entry and replacing it with entries expanding its children. Children then known to be uninteresting can be omitted, and variations in priority can be incorporated. This process can be repeated indefinitely, with more and more hoarding effort resulting in better and better approximations of the user's preferences.

5.1.4. Reference Spying

In many cases a user is not aware of the specific files accessed by an application. To facilitate construction of hoard profiles in such situations, Coda provides a `spy` program. This program can record all file references observed by Venus between a pair of start and stop events indicated by a user. Of course, different runtime behavior of the application can result in other files being accessed.

The `spy` program has been quite useful in deriving and tuning profiles. For example, it identified the reason why the X window system would sometimes hang when started from a disconnected workstation. It turns out that X font files are often stored in compressed format, with the X server expected to uncompress them as they are used. If the `uncompress` binary is not available when this occurs then the server will hang. Before `spy` was available, mysterious events such as this would happen in disconnected mode with annoying frequency. Since `spy`'s introduction we have been able to correct such problems on their first occurrence or, in many cases, avoid them altogether.

5.1.5. Periodic Hoard Walking

Background equilibration of the cache is an essential feature of hoarding. Without it there would be inadequate protection against involuntary disconnection. Even when voluntary disconnections are the primary type in an environment, periodic equilibration is still vital from a usability standpoint. First, it guards against a user who inadvertently forgets to demand a hoard walk before disconnecting. Second, it prevents a huge latency hit if and when a walk is demanded. This is very important because voluntary disconnections are often initiated when time is critical---for example, before leaving for the airport or when one is already late for dinner. Psychologically, users find it comforting that their machine is always "mostly current" with the state of the world, and that it can be made "completely current" with very little delay. Indeed, after a short break-in period with the system, users take for granted the fact that they'll be able to operate effectively if either voluntary or involuntary disconnection should occur.

5.1.6. Demand Hoard Walking

Foreground cache equilibration exists solely as an insurance mechanism for voluntary disconnections. The most common scenario for demand walking concerns a user who has been computing at their desktop workstation and is about to detach their laptop and take it home to continue work in the evening. In order to make sure that the latest versions of objects are cached, the user must force a hoard walk. An easy way to do this is to put the line "`hoard walk`" in one's `.logout` file. Most users, however, seem to like the reassurance of issuing the command manually, and internalize it as part of their standard shutdown

procedure. In any case, the requirement for demand walking before voluntary disconnection cannot be eliminated since the background walk period cannot be set too close to 0. This bit of non-transparency has not been a source of complaint from our users, but it could conceivably be a problem for a less sophisticated user community.

5.2. Server Emulation

Our qualitative evaluation of server emulation centers on two issues: transparency and cache misses.

5.2.1. Transparency

Server emulation by Venus has been quite successful in making disconnected operation transparent to users. Many involuntary disconnections have not been noticed at all, and for those that have the usual indication has been only a pause of a few seconds in the user's foreground task at reintegration time. Even with voluntary disconnections, which by definition involve explicit manual actions, the smoothness of the transition has generally caused the user's awareness of disconnection to fade quickly.

The high degree of transparency is directly attributable to our use of a single client agent to support both connected and disconnected operation. If, like FACE [1], we had used a design with separate agents and local data stores for connected and disconnected operation, then every transition between the two modes would have been visible to users. Such transitions would have entailed the substitution of different versions of the same logical objects, severely hurting transparency.

5.2.2. Cache Misses

Many disconnected sessions experienced by our users, including many sessions of extended duration, involved no cache misses whatsoever. We attribute this to two primary factors. First, as noted in the preceding subsection, hoarding has been a generally effective technique for our user population. Second, most of our disconnections were of the voluntary variety, and users typically embarked on those sessions with well-formed notions of the tasks they wanted to work on. For example, they took their laptop home with the intent of editing a particular paper or working on a particular software module; they did not normally disconnect with the thought of choosing among dozens of distinct tasks.

When disconnected misses did occur, they often were not fatal to the session. In most such cases the user was able to switch to another task for which the required objects were cached. Indeed, it was often possible for a user to "fall-back" on different tasks two or three times before they gave up and terminated the session. Although this is a result we expected, it was still quite a relief to observe it in practice. It confirmed our belief that hoarding need not be 100% effective in order for the system to be useful.

On a cache miss, the default behavior of Venus is to return an error code. A user may optionally request Venus to block processes until cache misses can be serviced. In our experience, users have made no real use of the blocking option for handling disconnected misses. We conjecture that this is due to the fact that all of our involuntary disconnections have occurred in the context of networks with high mean-time-to-repair (MTTR). We expect blocking will be a valuable and commonly used option in in networks with low MTTRs.

5.3. Reintegration

Our qualitative evaluation of reintegration centers on two issues: performance and failures.

5.3.1. Performance

The latency of reintegration has not been a limiting factor in our experience. Most reintegrations have taken less than a minute to complete, with the majority having been in the range of 5-20 seconds. Moreover, many reintegrations have been triggered by background Venus activity rather than new user requests, so the perceived latency has often been nil.

Something which we have not experienced but consider a potential problem is the phenomenon of a *reintegration storm*. Such a storm could arise when many clients try to reintegrate with the same server at about the same time. This could occur, for instance, following recovery of a server or repair of a major network artery. The result could be serious overloading of the server and greatly increased reintegration

times. We believe that we have not observed this phenomenon yet because our client population is too small and because most of our disconnections have been voluntary rather than the result of failures. We do, however, have two ideas on how the problem should be addressed:

- Have a server return a "busy" result once it reaches a threshold level of reintegration activity. Clients could back-off different amounts of time according to whether their reintegration was triggered by foreground or background activity, then retry. The back-off amounts in the foreground case would be relatively short and those in the background relatively long.
- Break operation logs into independent parts and reintegrate the parts separately. Of course, only the parts corresponding to foreground triggering should be reintegrated immediately; reintegration of the other parts should be delayed until the storm is over.

5.3.2. Detected Failures

Failed reintegrations have been very rare in our experience with Coda. The majority of failures that have occurred have been due to bugs in the implementation rather than update conflicts. We believe that this mostly reflects the low degree of write-sharing intrinsic to our environment. There is no doubt, however, that it also reflects certain behavioral adjustments on the part of our users. The most significant such adjustments were the tendencies to favor indirect over direct forms of sharing, and to avoid synchronization actions when one was disconnected. So, for example, if two users were working on the same paper or software module, they would be much more likely to each make their own copy and work on it than they would to make incremental updates to the original object. Moreover, the "installation" of a changed copy would likely be delayed until a user was certain he or she was connected. Of course, this basic pattern of sharing is the dominant one found in any Unix environment. The observation here is that it appeared to be even more common among our users than is otherwise the case.

Although detected failures have been rare, recovering from those that have occurred has been irksome. If reintegration fails, Venus writes out the operation log and related container files to a local file called a *closure*. A tool is provided for the user to inspect the contents of a closure, to compare it to the state at the AVSG, and to replay it selectively or in its entirety.

Our approach of forming closures and storing them at clients has several problems:

- there may not be enough free space at the client to store the closure. This is particularly true in the case of laptops, on which disk space is already precious.
- the recovery process is tied to a particular client. This can be annoying if a user ever uses more than one machine.
- interpreting closures and recovering data from them requires at least an intermediate level of system expertise. Moreover, even for expert users it can be difficult to determine exactly why some reintegrations failed.

The first two limitations could be addressed by migrating closures to servers rather than keeping them at clients. That strategy was, in fact, part of the original design for disconnected operation, and it continues to look like a worthwhile option.

We believe that the third problem can be addressed through a combination of techniques that reduce the number of failures that must be handled manually, and simplify the handling of those that remain. We discuss our current work in this area in Section 7.3.

5.4. Other Observations

5.4.1. Optimistic Replication

The decision to use optimistic rather than pessimistic replica control was undoubtedly the most fundamental one in the design of Coda. Having used the system for more than two years now, we remain convinced that the decision was the correct one for our type of environment.

Any pessimistic protocol must, in one way or another, allocate the rights to access objects when disconnected to particular clients. This allocation involves an unpleasant compromise between availability and ease of use. On the one hand, eliminating user involvement increases the system's responsibility, thereby lowering the sophistication of the allocation decisions. Bad allocation decisions translate directly

into lowered availability; a disconnected client either does not have a copy of a critical object, or has a copy that it cannot use because of insufficient rights. On the other hand, the more involved users are in the allocation process, the less transparent the system becomes.

An optimistic replication approach avoids the need to make a priori allocation decisions altogether. Our users have never been faced with the situation in which they are disconnected and have an object cached, but they cannot access it because of insufficient replica control rights. Similarly, they have never had to formally "grab control" of an object in anticipation of disconnection, nor have they had to "wrest control" from another client that had held rights they didn't really need. The absence of these situations has been a powerful factor in making the system effective and pleasant to use.

Of course, there is an advantage of pessimistic over optimistic replica control, which is that reintegration failures cannot occur. Our experience indicates that, in a Unix file system environment, this advantage is not worth much because there simply are very few failed reintegrations. The amount and nature of sharing in the workload make reintegration failures unlikely, and users adopt work habits that reduce their likelihood even further. In effect, the necessary degree of cross-partition synchronization is achieved voluntarily, rather than being enforced by a pessimistic algorithm.

Herlihy [3] once gave the following motivation for optimistic concurrency control, which applies equally well to optimistic replica control:

> ...[optimistic replica control] is based on the premise that it is more effective to apologize than to ask permission.

In our environment, the cases in which one would wrongfully be told "no" when asking permission vastly outnumber those in which a "no" would be justified. Hence, we have found it far better to suffer the occasional indignity of making an apology than the frequent penalty of a wrongful denial.

5.4.2. Security

There have been no detected violations of security in our use of Coda, and we believe that there have been no undetected violations either. The friendliness of our testbed environment is undoubtedly one important explanation for this. However, we believe that the Coda implementation would do well security-wise even under more hostile conditions.

The basis for this belief is the faithful emulation of the AFS security model. Coda servers demand to see a user's credentials on every client request, including reintegration. Credentials can be stolen, but this requires subversion of a client or a network-based attack. Network attacks can be thwarted through the use of (optional) message encryption, and the danger of stolen credentials is limited by associating fixed lifetimes with them. Access-control lists further limit the damage due to credential theft by confining it to areas of the namespace legitimately accessible to the subverted principal. Disconnected operation provides no back-doors that can be used to circumvent these controls.

AFS has provided good security at large-scale and under circumstances that are traditionally somewhat hostile. Indeed, we know of no other distributed file system in widespread use that provides better security with a comparable level of functionality. This strongly suggests that security would not be a factor limiting Coda's deployment beyond our testbed environment.

5.4.3. Public Workstations

Some computing environments include a number of public workstation clusters. Although it was never a primary goal to support disconnected operation in that domain, it was something that we hoped would be possible and which influenced Coda's early design to some degree.

Our experience with disconnected operation has convinced us that it is simply not well suited to public access conditions. One problem is that of security. Without disconnected operation, it is the case that when a user leaves a public workstation his or her data is all safely at servers and he or she is totally independent of that workstation. This allows careful users to flush their authentication tokens and their sensitive data from the cache when they depart, and to similarly "scrub" the workstation clean when they arrive. But with disconnected operation, scrubbing is not necessarily an option. The departing user cannot scrub if he or she has dirty objects in the cache, waiting to be reintegrated. The need to leave valid

authentication tokens with the cache manager is particularly worrying, as that exposes the user to arbitrary damage. And even if damage does not arise due to security breach, the departing user still must worry that a future user will scrub the machine and thereby lose his or her pending updates.

The other major factor that makes disconnected operation unsuited to public workstations is the latency associated with hoarding. Loading a cache with one's full "hoardable set" can take many minutes. Although this is done in the background, it can still slow a client machine down considerably. Moreover, if a user only intends to use a machine briefly, as is often the case with public machines, then the effort of hoarding is likely to be a waste. It is only when the cost of hoarding can be amortized over a long usage period that it becomes a worthwhile exercise.

6. Quantitative Evaluation

An earlier paper [7] presented measurements that shed light on key aspects of disconnected operation in Coda. Perhaps the most valuable of those measurements was the compelling evidence that optimistic replication in a Coda-like environment would indeed lead to very few write-write conflicts. That evidence was based on a year-long study of a 400-user AFS cell at CMU. The data showed that cross-user write-sharing was extremely rare. Over 99% of all file and directory modifications were by the previous writer, and the chances of two different users modifying the same object less than a day apart was at most 0.72%. If certain system administration files which skewed the data were excluded, the absence of write-sharing was even more striking: more than 99.7% of all mutations were by the previous writer, and the chances of two different users modifying the same object within a week were less than 0.3%.

In the following sections we present new measurements of Coda. These measurements either address questions not considered in our earlier paper, or provide more detailed and up-to-date data on issues previously addressed. The questions we address here are:

- How large a local disk does one need?
- How noticeable is reintegration?
- How important are optimizations to the operation log?

6.1. Methodology

To estimate disk space requirements we relied on simulations driven by an extensive set of file reference traces that we had collected [12]. Our analysis was comprehensive, taking into account the effect of all references in a trace whether they were to Coda, AFS or the local file system. The traces were carefully selected for sustained high levels of activity from over 1700 samples. We chose 10 workstation traces, 5 representing 12-hour workdays and the other 5 representing week-long activity. Table 1 identifies these traces and presents a summary of the key characteristics of each.

Trace Identifier	Machine Name	Machine Type	Simulation Start	Trace Records
Work-Day #1	brahms.coda.cs.cmu.edu	IBM RT-PC	25-Mar-91, 11:00	195289
Work-Day #2	holst.coda.cs.cmu.edu	DECstation 3100	22-Feb-91, 09:15	348589
Work-Day #3	ives.coda.cs.cmu.edu	DECstation 3100	05-Mar-91, 08:45	134497
Work-Day #4	mozart.coda.cs.cmu.edu	DECstation 3100	11-Mar-91, 11:45	238626
Work-Day#5	verdi.coda.cs.cmu.edu	DECstation 3100	21-Feb-91, 12:00	294211
Full-Week #1	concord.nectar.cs.cmu.edu	Sun 4/330	26-Jul-91, 11:41	3948544
Full-Week #2	holst.coda.cs.cmu.edu	DECstation 3100	18-Aug-91,23:21	3492335
Full-Week #3	ives.coda.cs.cmu.edu	DECstation 3100	03-May-91,12:15	4129775
Full-Week #4	messiaen.coda.cs.cmu.edu	DECstation 3100	27-Sep-91, 00:15	1613911
Full-Week #5	purcell.coda.cs.cmu.edu	DECstation 3100	21-Aug-91,14:47	2173191

Table 1: Vital Statistics for the Work-Day and Full-Week Traces

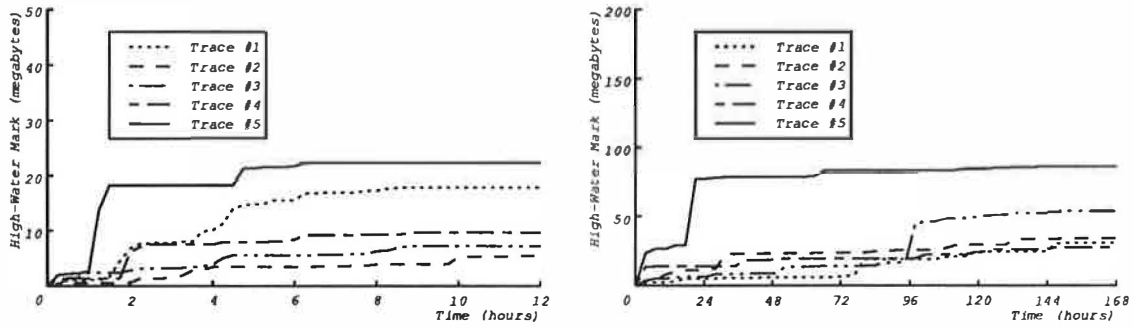
To address the question of reintegration latency, we performed a well-defined set of activities while disconnected and timed the duration of the reintegration phase after each. One of these activities was the running of the Andrew benchmark [4]. Another was the compilation of the then-current version of Venus.

A third class of activities corresponded to the set of traces in Table 1. We effectively "inverted" these traces and generated a command script from each. When executed, each of these scripts produced a trace isomorphic to the one it was generated from. This gave us a controlled and repeatable way of emulating real user activity.

We combined these two techniques to assess the value of log optimizations. Using our trace suite, we compared disk usage with and without optimizations enabled. We also measured the impact of log optimizations on reintegration latency for the set of activites described in the previous paragraph.

6.2. Disk Space Requirements

Figure 2 shows the *high-water mark* of cache space usage for the Work-Day and Full-Week traces as a function of time. The high-water mark is simply the maximum cache space in use at the current and all previous points in the simulation. The high-water mark therefore never declines, although the current cache space in use may (due to the deletion of objects).



(a) Work-Day Traces (b) Full-Week Traces

This graph presents the high-water marks of cache usage for each trace in Table 1. Note that the vertical axis on the graphs for Work-Day and Full-Week traces are different.

Figure 2: High-Water Marks of Cache Space Usage

These curves indicate that cache space usage tends to grow rapidly at the start, but tapers off quite soon. For example, most of the Work-Day traces had reached 80% of their 12-hour high-water marks within a few hours of their start. Similarly, all but one of the Full-Week traces had reached a substantial fraction of their 7-day high-water marks by the end of the second day. Note that it was not the case that the workstations simply became idle after the first parts of the traces; the traces were carefully selected to ensure that users were active right through to the end of the simulated periods.

These results are encouraging from the point of view of disconnected operation. The most expansive of the Work-Day traces peaked out below 25 MB, with the median of the traces peaking at around 10 MB. For the Full-Week traces, the maximum level reached was under 100 MB and the median was under 50 MB. This suggests that today's typical desktop workstation, with a disk of 100 MB to 1 GB, should be able to support many disconnections of a week or more in duration. Even the 60-200MB disk capacity of many laptops today is adequate for extended periods of disconnected operation. These observations corroborate our first-hand experience in using Coda laptops.

6.3. Reintegration Latency

Reintegration latency is a function of the update activity at a client while disconnected. In our use of the system, most one-day disconnections have resulted in reintegration times of a minute or less, and a few longer disconnections have taken a few minutes. Table 2 reports the latency, number of log records, and amount of data *back-fetched* for each of our reintegration experiments. Back-fetching refers to the transfer of data from client to server representing disconnected file store operations. Reintegration occurs in three subphases: a *prelude*, an *interlude*, and *postlude*. Latency is reported separately for the subphases as well as in total. On average, these subphases contributed 10%, 80% and 10% respectively to the total latency.

These results confirm our subjective experience that reintegration after a typical one-day disconnection is hardly perceptible. The Andrew benchmark, Venus make, and four of the five Work-Day trace-replay

Task	Log Record Total	Back-Fetch Total	Latency			
			Prelude	Interlude	Postlude	Total
Andrew Benchmark	203	1.2	1.8	7.5	.8	10 (1)
Venus Make	146	10.3	1.4	36.2	.4	38 (1)
Work-Day #1 Replay	1422	4.9	6.5	54.7	10.7	72 (5)
Work-Day #2 Replay	316	.9	1.9	9.8	1.7	14 (1)
Work-Day #3 Replay	212	.8	1.0	6.2	.9	8 (0)
Work-Day #4 Replay	873	1.3	2.9	23.2	5.9	32 (3)
Work-Day # 5 Replay	99	4.0	.9	20.5	.5	22 (2)
Full-Week #1 Replay	1802	15.9	15.2	138.8	21.9	176 (3)
Full-Week #2 Replay	1664	17.5	16.2	129.1	15.0	160 (2)
Full-Week #3 Replay	7199	23.7	152.6	881.3	183.0	1217 (12)
Full-Week #4 Replay	1159	15.1	5.1	77.4	7.0	90 (1)
Full-Week #5 Replay	2676	35.8	28.2	212.8	31.7	273 (9)

This data was obtained with a DECstation 5000/200 client and server. The Back-Fetch figures are in megabytes. Latency figures are in seconds. Each latency number is the mean of three trials. The numbers in parentheses in the "Latency Total" column are standard deviations. Standard deviations for the individual phases are omitted for space reasons.

Table 2: Reintegration Latency

experiments all reintegrated in under 40 seconds. The other Work-Day trace-replay experiment took only slightly more than a minute to reintegrate.

The reintegration times for the week-long traces are also consistent with our qualitative observations. Four of the five week-long trace-replay experiments reintegrated in under five minutes, with three completing in three minutes or less. The other trace-replay experiment is an outlier, requiring about 20 minutes to reintegrate.

In tracking down the reason for this anomaly, we discovered a significant shortcoming of our implementation. We found, much to our surprise, that the time for reintegration bore a non-linear relationship to the size of the operation log and the number of bytes back-fetched. Specifically, the regression coefficients were .026 for the number of log records, .0000186 for its square, and 2.535 for the number of megabytes back-fetched. The quality of fit was excellent, with an R^2 value of 0.999.

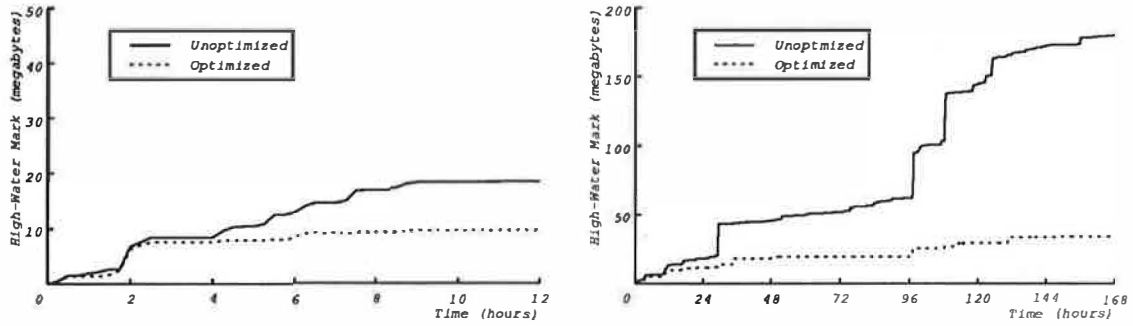
The first coefficient implies a direct overhead per log record of 26 milliseconds. This seems about right, given that many records will require at least one disk access at the server during the interlude phase. The third coefficient implies a rate of about 400 KB/s for bulk data transfer. This too seems about right, given that the maximum disk-to-disk transfer rate between 2 DECstation 5000/200s on an Ethernet that we've observed is 476 kilobytes/second. The source of the quadratic term turned out to be a naive sorting algorithm that was used on the servers to avoid deadlocks during replay. For disconnected sessions of less than a week, the linear terms dominate the quadratic term. This explains why we have never observed long reintegration times in normal use of Coda. But around a week, the quadratic term begins to dominate. Clearly some implementation changes will be necessary to make reintegration linear. We do not see these as being conceptually difficult, but they will require a fair amount of code modification.

It is worth making two additional points about reintegration latency here. First, because reintegration is often triggered by a daemon rather than a user request, perceived latency is often nil. That is, reintegrations often occur entirely in the background and do not delay user computation at all. Second, the trace-replay experiments reflect activity that was originally performed in a number of volumes. For the Work-Day traces 5-10 volumes were typically involved, and for the Full-Week traces the number was typically 10-15. For logistical reasons, the replay experiments were each performed within a single Coda volume. Hence, there was only one reintegration for each experiment. Following an actual disconnected execution of the trace activity, though, there would have been a number of smaller reintegrations instead of one large one. If the reintegrated volumes were spread over different servers, a significant amount of parallelism could have been realized. The total latency might therefore have been much smaller, perhaps by a factor of three or four.

6.4. Value of Log Optimizations

Venus uses a number of optimizations to reduce the length of the operation log. A small log conserves disk space, a critical resource during periods of disconnection. It also improves reintegration performance by reducing latency and server load. Details of these optimizations can be found elsewhere [7, 8].

In order to understand how much space these optimizations save in practice, our Venus simulator was augmented to report cache usage statistics with the optimizations turned off as well as on. Figure 3 compares the median high-water marks of space usage for our trace suite with and without optimizations.



(a) Work-Day Traces (b) Full-Week Traces

Each curve above represents the median values of the high-water marks of space usage for the five corresponding traces. Note that the vertical axis on the two graphs are different.

Figure 3: Optimized versus Unoptimized Cache Space High-Water Marks

The differences between the curves in each case are substantial. After an initial period in which the two curves increase more or less together, the unoptimized curves continue to increase while the optimized curves taper off. For the Work-Day traces, the unoptimized total has grown to nearly twice that of the optimized case by the 12-hour mark. The trend continues unabated with the Full-Week traces, with the unoptimized total being more than 5 times that of the optimized case at the end of the week. This equates to a difference of more than 145 megabytes. The slopes of the two lines indicate that the difference would increase even further over periods of greater length.

Table 3 shows that the differences for certain individual traces are even more striking. That table lists the unoptimized and optimized totals for each trace at its termination. In addition, each total is broken down into its two constituents: cache container space and RVM space. Cache container space refers to the space used by the local files that are used to hold the images of the current versions of Coda files.

The greatest savings tend to be realized in cache container space, although the RVM space savings can also be substantial. The far right column shows the ratio of unoptimized to optimized total space usage. The maximum ratio for the Work-Day traces is 3.1, indicating that more than three times the amount of space would have been needed without the optimizations. The maximum ratio for the Full-Week traces is an astonishing 28.9, which corresponds to a difference of more than 850 megabytes.

Trace	Container Space		RVM Space		Total Space		
	Unopt	Opt	Unopt	Opt	Unopt	Opt	Ratio
Work-Day #1	34.6	15.2	2.9	2.7	37.5	17.8	2.1
Work-Day #2	14.9	4.0	2.3	1.5	17.2	5.5	3.1
Work-Day #3	7.9	5.8	1.6	1.4	9.5	7.2	1.3
Work-Day #4	16.7	8.2	1.9	1.5	18.6	9.7	1.9
Work-Day #5	59.2	21.3	1.2	1.1	60.4	22.3	2.7
Full-Week #1	872.6	25.9	11.7	4.8	884.3	30.6	28.9
Full-Week #2	90.7	28.3	13.3	5.9	104.0	34.2	3.0
Full-Week #3	119.9	45.0	46.2	9.1	165.9	54.0	3.1
Full-Week #4	222.1	23.9	5.5	3.9	227.5	27.7	8.2
Full-Week #5	170.8	79.0	9.1	7.7	179.8	86.5	2.1

The figures in the "Unopt" and "Opt" columns are in megabytes

Table 3: Optimized versus Unoptimized Space Usage

These results confirm that log optimizations are critical for managing space at a disconnected client. But they are also important for keeping reintegration latency low. To confirm this, we used the regression results and measured values of unoptimized log records and data back-fetched from the experiments reported in Section 6.3. Using this information we estimated how long reintegration would have taken, had log optimizations not been done. Table 4 presents our results.

Task	Log Record Total		Back-Fetch Total		Latency		
	Unopt	Opt	Unopt	Opt	Unopt	Opt	Ratio
Andrew Benchmark	211	203	1.5	1.2	10	10	1.0
Venus Make	156	146	19.5	10.3	54	38	1.4
Work-Day #1 Replay	2422	1422	19.1	4.9	221	72	3.1
Work-Day #2 Replay	4093	316	10.9	.9	446	14	31.9
Work-Day #3 Replay	842	212	2.1	.8	41	8	5.1
Work-Day #4 Replay	2439	873	8.5	1.3	196	32	6.1
Work-Day #5 Replay	545	99	40.9	4.0	123	22	5.6
Full-Week #1 Replay	33923	1802	846.9	15.9	24433	176	138.8
Full-Week #2 Replay	36855	1664	62.4	17.5	26381	160	164.9
Full-Week #3 Replay	175392	7199	75.0	23.7	576930	1217	474.1
Full-Week #4 Replay	8519	1159	199.1	15.1	2076	90	23.1
Full-Week #5 Replay	8873	2676	92.7	35.8	1930	273	7.1

Back-fetch figures are in megabytes, and latencies in seconds. The reported latencies are the means of three trials. Standard deviations are omitted for brevity.

Table 4: Optimized versus Unoptimized Reintegration Latency

The time savings due to optimizations are enormous. The figures indicate that without the optimizations, reintegration of the trace-replay experiments would have averaged 10 times longer than actually occurred for the Work-Day set, and 160 times longer for the Full-Week set. Reintegrating the unoptimized replay of Full-Week trace #3 would have taken more than 6 days, or nearly as long as the period of disconnection! Obviously, much of the extra time is due to the fact that the unoptimized log record totals are well into the range at which the quadratic steps of our implementation dominate. Although the savings will not be as great when our code is made more efficient, it will not be inconsequential by any means. Even if the quadratic term is ignored, the ratios of unoptimized to optimized latency are still pronounced: on average, 4.5 for Work-Day traces and 7.6 for the Full-Week traces.

7. Work in Progress

Coda is a system under active development. In the following sections we describe work currently under way to enhance the functionality of Coda as well as to alleviate some of its current shortcomings.

7.1. Exploiting Weak Connectivity

Although disconnected operation in Coda has proven to be effective for using distributed file systems from mobile computers, it has several limitations:

- cache misses are not transparent. A user may be able to work in spite of some cache misses, but certain critical misses may frustrate these efforts.
- longer disconnections increase the likelihood of resource exhaustion on the client from the growing operation log and new data in the cache.
- longer disconnections also increase the probability of conflicts requiring manual intervention upon reconnection.

Wireless technologies such as cellular phone and even traditional dialup lines present an opportunity to alleviate some of the shortcomings of disconnected operation. These *weak connections* are slower than LANs, and some of the wireless technologies have the additional properties of intermittence and non-trivial cost. The characteristics of these networks differ substantially from those of LANs, on which many distributed file systems are based.

We are exploring techniques to exploit weak connectivity in a number of ways:

- Coda clients will manage use of the network intelligently. By using the network in a preemptive, prioritized fashion, the system will be able to promptly service critical cache misses. It will propagate mutations back to the servers in the background to prevent conflicts that would arise at reintegration time and to reclaim local resources. It will allow users to weaken consistency on an object-specific basis to save bandwidth.
- Coda clients will minimize bandwidth requirements by using techniques such as batching and compression. Techniques in this class demand more server computation per request, so the state of the server will play a role in the use of these techniques.
- Coda clients will dynamically detect and adapt to changes in network performance. This will be especially important when connectivity is intermittent.

An important consideration in the use of weak connections is the issue of callback maintenance. Callback-based cache consistency schemes were designed to minimize client-server communication, but with an underlying assumption that the network is fast and reliable. After a network failure all callbacks are invalid. In an intermittent low-bandwidth network, the cost of revalidation may be substantial and may nullify the performance benefits of callback-based caching.

To address this issue we have introduced the concept of *large granularity callbacks*. A large granularity trades off precision of invalidation for speed of validation after connectivity changes. Venus will choose the granularity on a per-volume basis, adapting to the current networking conditions as well as the observed rate of callback breaks due to mutations elsewhere in the system. Further details on this approach can be found in a recent paper [11].

7.2. Hoarding Improvements

We are in the process of developing tools and techniques to reduce the burden of hoarding on users, and to assist them in accurately assessing which files to hoard. A key problem we are addressing in this context is the choice of proper metrics for evaluating the quality of hoarding.

Today, the only metric of caching quality is the *miss ratio*. The underlying assumption of this metric is that all cache misses are equivalent (that is, all cache misses exact roughly the same penalty from the user). This assumption is valid in the absense of disconnections and weak connections because the performance penalty resulting from a cache miss is small and independent of file length. This assumption is not valid during disconnected operation and may not be valid for weakly-connected operation, depending on the strength of the connection. The cache miss ratio further assumes that the timing of cache misses is irrelevant. But the user may react differently to a cache miss occurring within the first few minutes of disconnection than to one occurring near the end of the disconnection.

We are extending the analysis of hoarding tools and techniques using new metrics such as:

- the time until the first cache miss occurs.
- the time until a critical cache miss occurs.
- the time until the cumulative effect of multiple cache misses exceeds a threshold.
- the time until connection transparency is lost.
- the percentage of the cache actually referenced when disconnected or weakly-connected, as a measure of overly-generous hoarding.
- the change in connected-mode miss ratio due to hoarding.

We plan to use these metrics to evaluate the relative value of different kinds of hoarding assistance. For example, under what circumstances does one tool prove better than another? Some of our experiments will be performed on-line as users work. Others will be performed off-line using post-mortem analysis of file reference traces. The tools we plan to build will address a variety of needs pertinent to hoarding. Examples include tools to support task-based hoarding and a graphical interface to accept hoard information and provide feedback regarding the cache contents.

7.3. Application-Specific Conflict Resolution

Our experience with Coda has established the importance of optimistic replication for mobile computing. But optimistic replication brings with it the need to detect and resolve concurrent updates in multiple partitions. Today Coda provides for transparent resolution of directory updates. We are extending our work to support transparent resolution on arbitrary files. Since the operating system does not possess any semantic knowledge of file contents, it is necessary to obtain assistance from applications.

The key is to provide an application-independent invocation mechanism that allows pre-installed, *application-specific resolvers (ASRs)* to be transparently invoked and executed when a conflict is detected. As a practical example of this approach, consider a calendar management application. The ASR in this case might merge appointment database copies by selecting all non-conflicting appointments and, for those time slots with conflicts, choosing to retain one arbitrarily and sending mail to the rejected party(s).

We have recently described such an interface for supporting ASRs [10]. Our design addresses the following issues:

- An application-independent interface for transparently invoking ASRs.
- An inheritance mechanism to allow convenient rule-based specification of ASRs based on attributes such as file extension or position in the naming hierarchy.
- A fault tolerance mechanism that encapsulates ASR execution.

Even in situations where manual intervention is unavoidable, ASR technology may be used for partial automation. Consider, for example, the case of two users who have both edited a document or program source file. An "interactive ASR" could be employed in this case which pops up side-by-side windows containing the two versions and highlights the sections which differ. The user could then quickly perform the merge by cutting and pasting. Similarly, a more useful version of the calendar management ASR might begin with a view of the appointment schedule merged with respect to all non-conflicting time slots, then prompt the user to choose between the alternatives for each slot that conflicts.

Another class of ASRs that may be valuable involves automatic re-execution of rejected computations by Venus. This is precisely the approach advocated by Davidson in her seminal work on optimistic replication in databases [2], and it will be feasible to use in Coda once the transactional extensions described in Section 7.4 are completed. Automatic re-execution would be appropriate in many cases involving the `make` program, for example.

7.4. Transactional Extensions for Mobile Computing

With the increasing frequency and scale of data sharing activities made possible by distributed Unix file systems such as AFS, Coda, and NFS [13], there is a growing need for effective consistency support for concurrent file accesses. The problem is especially acute in the case of mobile computing, because extended periods of disconnected or weakly-connected operation may increase the probability of read-write inconsistencies in shared data.

Consider, for example, a CEO using a disconnected laptop to work on a report for an upcoming shareholder's meeting. Before disconnection she cached a spreadsheet with the most recent budget figures available. She writes her report based on the numbers in that spreadsheet. During her absence, new budget figures become available and the server's copy of the spreadsheet is updated. When the CEO returns and reintegrates, she needs to discover that her report is based on stale budget data. Note that this is not a write-write conflict, since no one else has updated her report. Rather it is a read-write conflict, between the spreadsheet and the report. No Unix system today has the ability to detect and deal with such problems.

We are exploring techniques for extending the Unix interface with transactions to provide this functionality. A key attribute of our effort is upward compatibility with the Unix paradigm. Direct transplantation of traditional database transactions into Unix is inappropriate. The significant differences in user environment, transaction duration and object size between Unix file systems and database systems requires transactional mechanisms to be specially tailored. These considerations are central to our design of a new kind of transaction, called *isolation-only transaction*, whose use will improve the consistency properties of Unix file access in partitioned networking environments.

A distinct but related area of investigation is to explore the effects of out-of-band communication on mobile computing. For example, a disconnected user may receive information via a fax or phone call that he incorporates into the documents he is working on. What system support can we provide him to demarcate work done before that out-of-band communication? This will become important if he later needs to extricate himself from a write-write or read-write conflict.

8. Conclusion

In this paper, we have focused on disconnected operation almost to the exclusion of server replication. This is primarily because disconnected operation is the newer concept, and because it is so central to solving the problems that arise in mobile computing. However, the importance of server replication should not be underestimated. Server replication is important because it reduces the frequency and duration of disconnected operation. Thus server replication and disconnected operation are properly viewed as complementary mechanisms for high availability.

Since our original description of disconnected operation in Coda [7] there has been considerable interest in incorporating this idea into other systems. One example is the work by Huston and Honeyman [5] in implementing disconnected operation in AFS. These efforts, together with our own substantial experience with disconnected operation in Coda, are evidence of the soundness of the underlying concept and the feasibility of its effective implementation.

None of the shortcomings exposed in over two years of serious use of disconnected operation in Coda are fatal. Rather, they all point to desirable ways in which the system should evolve. We are actively refining the system along these dimensions, and have every reason to believe that these refinements will render Coda an even more usable and effective platform for mobile computing.

Acknowledgments

We wish to thank all the members of the Coda project, past and present, for their contributions to this work. David Steere, Brian Noble, Hank Mashburn, and Josh Raiff deserve special mention. We also wish to thank our brave and tolerant user community for their willingness to use an experimental system. This work was supported by the Advanced Research Projects Agency (Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division(AFSC), U.S. Air Force, Wright-Patterson AFB under Contract F33615-90-C-1465, Arpa Order No. 7597), the National Science Foundation (Grant ECD 8907068), IBM, Digital Equipment Corporation, and Bellcore.

References

[1] Cova, L.L.
Resource Management in Federated Computing Environments.
PhD thesis, Department of Computer Science, Princeton University, October, 1990.

[2] Davidson, S.
Optimism and Consistency in Partitioned Distributed Database Systems.
ACM Transactions on Database Systems 3(9), September, 1984.

[3] Herlihy, M.
Optimistic Concurrency Control for Abstract Data Types.
In *Proceedings of the Fifth Annual Symposium on Principles of Distributed Computing.* August, 1986.

[4] Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J.
Scale and Performance in a Distributed File System.
ACM Transactions on Computer Systems 6(1), February, 1988.

[5] Huston, L., Honeyman, P.
Disconnected Operation for AFS.
In *Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing.* Cambridge, MA, August, 1993.

[6] Kazar, M.L. .
Synchronization and Caching Issues in the Andrew File System.
In *Winter Usenix Conference Proceedings, Dallas, TX.* 1988.

[7] Kistler, J.J., Satyanarayanan, M.
Disconnected Operation in the Coda File System.
ACM Transactions on Computer Systems 10(1), February, 1992.

[8] Kistler, J.J.
Disconnected Operation in a Distributed File System.
PhD thesis, Department of Computer Science, Carnegie Mellon University, May, 1993.

[9] Kumar, P., Satyanarayanan, M.
Log-Based Directory Resolution in the Coda File System.
In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems.* San Diego, CA, January, 1993.

[10] Kumar, P., Satyanarayanan, M.
Supporting Application-Specific Resolution in an Optimistically Replicated File System.
June, 1993.
submitted to 4th IEEE Workshop on Workstation Operating Systems, Napa, CA, October 1993.

[11] Mummert, L.B., Satyanarayanan, M.
File Cache Consistency in a Weakly Connected Environment.
June, 1993.
submitted to 4th IEEE Workshop on Workstation Operating Systems, Napa, CA, October 1993.

[12] Mummert, L.B.
Efficient Long-Term File Reference Tracing.
1993.
in preparation.

[13] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B.
Design and Implementation of the Sun Network Filesystem.
In *Summer Usenix Conference Proceedings.* 1985.

[14] Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.
Coda: A Highly Available File System for a Distributed Workstation Environment.
IEEE Transactions on Computers 39(4), April, 1990.

[15] Satyanarayanan, M.
Scalable, Secure, and Highly Available Distributed File Access.
IEEE Computer 23(5), May, 1990.

[16] Satyanarayanan, M., Mashburn, H.H., Kumar, P., Steere, D.C., Kistler, J.J.
Lightweight Recoverable Virtual Memory.
Technical Report CMU-CS-93-143, School of Computer Science, Carnegie Mellon University, March, 1993.

[17] Steere, D.C., Kistler, J.J., Satyanarayanan, M.
Efficient User-Level Cache File Management on the Sun Vnode Interface.
In *Summer Usenix Conference Proceedings, Anaheim.* June, 1990.

Using Prospero to Support Integrated Location-Independent Computing

B. Clifford Neuman Steven Seger Augart Shantaprasad Upasani

Information Sciences Institute
University of Southern California

Abstract

As computers become pervasive, users will access processing, storage, and communication resources from locations that have not been practical in the past. Such users will demand support for location-independent computing. While the basic system components used might change as the user moves from place to place, the appearance of the system should remain constant.

In this paper we discuss the role of, and requirements for, directory services in support of integrated, location-independent computing. We focus on two specific problems: the server selection problem and the user location problem. We present solutions to these problems based on the Prospero Directory Service. The solutions demonstrate several unique features of Prospero that make it particularly suited for support of location-independent computing.

1 Introduction

As the use of computers becomes pervasive the distinction between computer networks and computer systems will blur. In the ideal world, users will think of a computer network and the systems connected to it as a single system, rather than as a collection of systems connected by networks. Users will not want to use a different system each time they change their location. Although users will want to see a single system, they won't want to see the same system as every other user. Each user will want a system that is tailored to his or her particular needs.

This paper begins with a discussion of the characteristics of and requirements for what we call pervasive computing. We examine two problems that arise in such systems, the server selection problem and the user location problem, discussing the role played by a distributed directory service in their solution. In so doing, we describe the Prospero Directory Service, highlighting important features, and describing how it can be used to solve these problems.

2 Pervasive Computing

Pervasive computing combines aspects of ubiquitous computing with the integration of information and resources from many sources, within a single system tailored to the needs of a particular user. Whereas the focus of ubiquitous computing has been on the devices and the communication infrastructure, allowing the use of large and small computing devices from many locations, the focus of pervasive computing is on mechanisms that allow the pieces to be tied together to form a coherent whole. The two areas are not disjoint; each includes the other, only the perspective is different.

There are several characteristics to pervasive computing that place new demands on system organization and structure. One of the primary characteristics is mobility. The term mobility applies even to systems that don't support wireless communication; it is the mobility of users that is critical to pervasive computing. Users interact with the system from more than one location. We already see this on large university campuses where students log in from public terminal clusters.

The use of portable computers while traveling provides another example. In the future, users will be able to interact with the system through whatever I/O device is within reach as they travel from location to location.

A second characteristic of pervasive computing is scale. The number of objects and services to be managed can easily overwhelm the user, the geographic expanse of the system adds constraints to be considered when selecting servers, and the lack of a single organization that controls the system makes organization of these resources difficult. These characteristics can be addressed in part through support for customization. Users should be able to choose the resources and objects of interest, and treat the selected resources as a single system [5].

The directory service will play a critical role tying together the components of future systems. The requirements for such a directory service are greatly affected by the scale of the system, and the mobility of its users. One of the biggest problems to be addressed is support for transient information. The transient information in such a system comes in two forms: first, the choice of servers for certain operations may change as the user moves from location to location; and second, information about users and other mobile objects needs to be maintained.

3 The Server Selection Problem

We begin our discussion by considering the server selection problem. Selecting resources for use in a centralized system is straightforward: users choose from among the resources available and select defaults which rarely change. In traditional distributed systems, the selected resources are then located through remote name maps or directory services such as Sun's Network Information Services (formerly known as Yellow Pages) [6] and Hesiod [3].

The mobility of users complicates server selection; the user's choice of resources will often vary according to location. For example, while at home a user might want to use a printer at home and while at work, one down the hall. While traveling the user might want output faxed to the hotel's front desk, but only if there is no per-page charge for incoming faxes. Alternatively, a user might want output sent to the printer at home so that it is waiting upon return.

It should be possible for users to specify defaults in such a way that the binding is determined dynamically, when the service is needed, and based on a combination of user specified factors (e.g., cost and reliability), application requirements (e.g., support for PostScript), and transient factors (e.g., load and proximity).

3.1 Using Prospero to Solve the Server Selection Problem

With Prospero, users define a virtual system which, among other things, specifies the mapping of names to servers. This mapping is used to select the servers used by applications. Figure 1 shows how a virtual system is represented using Prospero. In the figure, the link labeled ROOT is a reference to the root of the user's file system through which the user sees the same files regardless of the location from which logged in. The Prospero File System is described elsewhere [4]. The directory referenced by the SESSIONS link identifies the locations from which the user is logged in and is described in Section 4.

In this virtual system, the server selection criteria are encoded in the CONFIG/SERVERS directory. In a traditional directory service, such a mapping would not provide the flexibility that is needed for pervasive computing. However, several features of the Prospero Directory Service allow the mapping to be determined dynamically. These features include virtual system aliases, union links, and filters.

3.1.1 Virtual system aliases

Prospero maintains several virtual system aliases that provide well defined starting points from which names may be resolved. These aliases end with the string #: and identify the virtual systems associated with the user, the home processor for the current login session[1] (the session), the processor on which an application is running (the platform), and open file descriptors.

[1] The virtual system for the session is associated with the workstation or I/O device through which the user is interacting with the system.

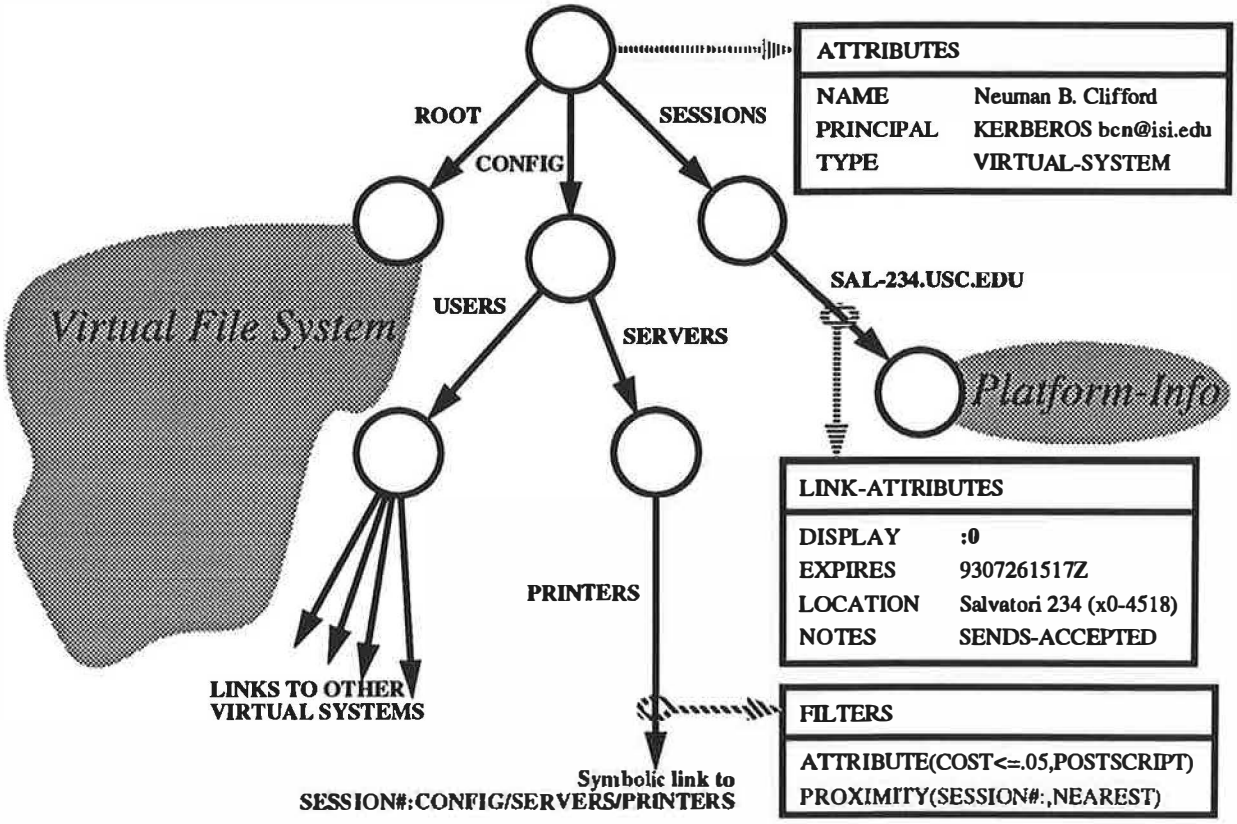


Figure 1: Structure of a Virtual System

In the figure, the directory CONFIG/SERVERS/PRINTERS defines the printers available to the user. If the user wanted to use the same printers always, the link would refer to a directory that explicitly named the printers to be used. Here, the user chose to define the available printers as a symbolic link to the directory of printers for the session. Because the target of the symbolic link begins with the string SESSION#: the rest of the name is resolved in the virtual system for the session, which is reachable through the node for the login platform and is part of the cloud labeled *Platform-Info* in the figure.

3.1.2 Union links

A user who wanted to choose from among the printer at home, the one at work, and those nearby, could use a union link to create a dynamic view of available printers. The CONFIG/SERVERS/PRINTERS link would refer to a directory with references to the printers at the user's home and office. That directory would include a symbolic union link to the directory of printers for the session. The resulting directory would appear to contain the union of the two directories.

3.1.3 Filters

Users should be cautious when allowing the system to select servers for use in unfamiliar locations. Business travelers have been taught this lesson by unscrupulous alternative telephone operator services that charge exorbitant fees for long distance phone calls. Using Prospero, users are able to constrain the selection of servers by applying a filter to a directory. A filter is a function attached to a link that modifies the result of a directory query. In figure 1 the user applied the **attribute()** filter to restrict the selection of printers to those charging no more than 5 cents per page, and supporting

PostScript. Of the printers selected by the **attribute()** filter, the **proximity()** filter selects the closest, provided that the implementer of the **proximity()** filter devised an appropriate heuristic.

A filter can also change the order in which servers are presented to the user setting the COLLATION-ORDER attribute of the links it returns. This allows filters to sort servers according to user specified criteria.

Prospero supports two kinds of filters: loadable and predefined. Loadable filters are dynamically loaded and executed during name resolution (their present implementation is not portable and presents security problems). Predefined filters are compiled into the name resolution library. Predefined filters have registered names and must be present in the name resolver (or on the server for some filters). Predefined filters must be widely supported to be useful; therefore they usually provide general operations such as selection based on the values of attributes.

3.1.4 Supporting mobile platforms

Our discussion so far has assumed that a stable platform exists for each session, and that a server directory has been defined for each platform. The discussion has ignored problems related to mobile platforms. The mobile platform problem can be addressed by another layer of indirection: the platform might itself use a filter in the definition of its server directory. Such a filter might locally broadcast a query in search of nearby platforms willing to provide such a directory. The resulting directory would include links for each server that responds.

3.1.5 Presenting selections to the user

When selecting a server, an application can indicate additional constraints to be applied by specifying additional filters. Once all filters have been applied, if the result is a single link, the referenced server can be used. For example, the NEAREST argument to the **proximity()** filter in the figure might result in the automatic selection of the nearest printer.

If the directory contains multiple links after the application of all filters, then the user could be prompted through a dialog box to select one. This might occur if the argument to the **proximity()** filter were specified as NEAREST 5, resulting in the return of the 5 nearest printers. Once selected by the user, the choice should remain in effect until some event specified by the user, such as a change in location, or a change in the list of available servers.

4 The User Location Problem

The user location problem is a second problem that illustrates the requirements imposed on a directory service by user mobility. In centralized systems, locating an active user is easy; users are either logged in, or they aren't. If logged in, the system records the the terminal in use and makes this information available to applications such as **finger**, **write**, and **send**. In a distributed system, the problem is considerably more complex.

A common approach to the user location problem is to replicate the data on all hosts. This is the approach taken by **rwho**. Systems broadcast the names of the users that are logged in, and others store this data locally where it can be searched by the user or application. The primary drawback of this approach is that it doesn't scale very well. A second concern is privacy; users might not want others to know where they are logged in, or that they are logged in at all.

A different approach is taken by the Zephyr [2] system at MIT's Project Athena [1]. Zephyr provides a single database that may be consulted when a user's location is needed. This database is replicated for reliability (technically, Zephyr provides a notification service that relies on this database, but it is the database that is of interest here). Zephyr addresses privacy because each user decides whether to register a session with Zephyr, and to what classes of other users the login location is to be visible. Zephyr does not provide fine-grained control over access to user location data. Though suitable for a large campus, the use of Zephyr as a user location database does not scale across administrative domains.

4.1 Using Prospero to Solve the User Location Problem

A third approach is to use a directory server to store user location information. Such a directory server would have to tolerate frequent updates. If a user is to be able to specify the principals who can obtain his or her location, then support for fine-grained access control is also necessary. Finally, it must be possible to authenticate both updates and queries.

The Prospero Directory Service already maintains information about a user's virtual system. This information has been extended to include information about login sessions. At login, an entry is added to a list of sessions, and at logout the entry is removed. By associating an expiration time with the entry, it is possible to detect sessions that are not properly terminated. By placing an access control list on the list of sessions, a user can specify on a per-principal basis the individuals to which the session is visible. The SESSIONS link in figure 1 points to the directory that maintains a list of sessions.

Prospero is a distributed directory service, and it is likely that user information will be distributed across a large number of systems, maintained by multiple organizations. Each directory server enforces its own access control. As such, if a user's directory information is stored on a trusted server for the user's organization, the user's privacy depends only upon the security of that server.

Through its support for customization, Prospero allows a user to define a set of colleagues, and the name used to refer to each. This set initially contains the names of other users in the local organization, with users beyond the organization named hierarchically based on the name of the organization to which they belong. Users can define their own short names for remote colleagues, after which they are referred to no differently than if they were local. This is represented in figure 1 by the CONFIG/USERS directory. In fact, this customization mechanism allows a user to define the set of colleagues considered local for use by `finger`; when run with no arguments it displays the locations of users currently active and identified as colleagues by the user.

When using modified versions of commands such as `send`, `talk`, or `finger`, the name of the target (another user) is specified relative to the CONFIG/USERS directory. The command consults the directory server to determine the location of the target user, and if found, performs the requested operation.

5 Establishing a Session

The solutions to both the server selection and the user location problems depend on several operations being performed when a new login session is established. This section describes what happens when a user logs into a system supporting the Prospero login program. In this discussion, the platform is the processor on which the application (in this case the login program) runs.

When a user attempts to log in to a system running the Prospero login, the system uses the name of the user to find the user's virtual system. For local users the virtual system is found in the directory of virtual systems for the local site. For remote users the virtual system is found starting from a directory that identifies other sites. The PRINCIPAL attribute associated with the user's virtual system is examined and used to select an appropriate authentication method. The user is authenticated and the login program determines whether the user is authorized to use the resources of the platform.

Once logged in, the namespace is defined by the user's virtual system and the virtual system alias SESSION#: is defined as the virtual system of the platform on which the user logged in. Next, if the user was suitably authenticated, an entry is made in the SESSIONS directory of the user's virtual system recording the platform, the login time, an expiration time, and other information associated with the session. During a session, name resolution occurs in the name space defined by the user's virtual system. When the session is terminated, the entry in the SESSIONS directory is removed.

6 Status

The Prospero Directory Service has been available since December 1990 and has been used to support the integration of information services on the Internet. The recent release of Version 5 of Prospero allows it to be more easily integrated with other applications. This paper described some of the ways that Prospero can be used to support integrated location-independent computing. Prospero presently provides the basic mechanism needed to address the problems discussed in this paper. We have started work on the user location problem as part of the implementation of a Prospero-based login program. We have not yet started work on the server selection problem, but plan to do so in the near future. To find out more about Prospero, or for directions on retrieving the latest distribution, send a message to info-prospero@isi.edu.

7 Summary

Pervasive computing places new demands on directory services. Among the demands is a need to support transient data, fine-grained authorization for queries and updates, and the ability to support dynamic (functional) bindings from names to servers and objects. These requirements are met by the Prospero Directory Service, which can play a role in support for truly integrated, location-independent computing.

Acknowledgments

Many individuals contributed to the design and implementation of Prospero. Ed Lazowska, John Zahorjan, David Notkin, Hank Levy, and Alfred Spector helped refine the ideas that ultimately led to the development of Prospero. Kwynn Buess, Steve Cliffe, Alan Emtage, George Ferguson, Bill Griswold, Sanjay Joshi, Brendan Kehoe, and Dan King helped with the implementation of Prospero and Prospero-based applications. Celeste Anderson, Sio-Man Cheang, Gennady Medvinsky, Santosh Rao, Eve Schooler, and Stuart Stubblebine commented on drafts of this paper.

References

[1] George A. Champine, Daniel E. Geer Jr., and William N. Ruh. Project Athena as a distributed computer system. *IEEE Computer*, 23(9):40–51, September 1990.

[2] C. Anthony DellaFera, Mark W. Eichin, Robert S. French, David C Jedlinsky, John T. Kohl, and William E. Sommerfeld. The Zephyr notification service. In *Proceedings of the Winter 1988 Usenix Conference*, pages 213–219, February 1988.

[3] Stephen P. Dyer. The Hesiod name server. In *Proceedings of the Winter 1988 Usenix Conference*, pages 183–189, February 1988.

[4] B. Clifford Neuman. The Prospero File System: A global file system based on the Virtual System Model. *Computing Systems*, 5(4):407–432, Fall 1992.

[5] B. Clifford Neuman. *The Virtual System Model: A Scalable Approach to Organizing Large Systems*. PhD thesis, University of Washington, June 1992. Department of Computer Science and Engineering Technical Report 92-06-04.

[6] Sun Microsystems. *Yellow Pages Protocol Specification*, February 1986. In Networking on the Sun Workstation.

This research was supported in part by the National Science Foundation (Grant No. CCR-8619663), the Washington Technology Centers, Digital Equipment Corporation, and the Advanced Research Projects Agency under NASA Cooperative Agreement NCC-2-539. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any of the funding agencies. Figures and descriptions in this paper were provided by the authors and are used with permission. The authors may be reached at USC/ISI, 4676 Admiralty Way, Marina del Rey, CA 90292-6695, USA. Telephone +1 (310) 822-1511, email bcn@isi.edu, swa@isi.edu, prasad@isi.edu.

The Qualcomm CDMA Digital Cellular System

Phil Karn
Qualcomm, Inc
10555 Sorrento Valley Rd
San Diego, CA 92121
karn@qualcomm.com

Introduction

Qualcomm, Inc. has developed and tested a digital spread spectrum cellular telephone system that will supplant and eventually replace the existing analog FM cellular system. TIA has recently standardized this "CDMA" (Code Division Multiple Access) as TIA IS-95, and several cellular carriers have already announced plans to deploy the service.

The original impetus for CDMA's development was the enormous popularity of existing narrow band analog FM cellular telephone systems. There is only so much radio spectrum available to deal with the flourishing demand for cellular service, and severe congestion already exists in several major service areas. A IS-95 system can carry about 10-15 times as many voice calls as an analog system in the same spectrum.

Although Qualcomm specifically designed IS-95 for voice telephony, it is easily adapted to packet data transmission. It can provide a service comparable to those of several existing and proposed metropolitan area wireless data networks with better performance, greater reliability, and lower cost.

A Note on Terminology

The term CDMA ("Code Division Multiple Access") is generic. It applies to any spread-spectrum multiple access system where different spreading codes (or different portions of the same spreading code) allow receivers to discriminate between multiple transmitters using the same frequency channel at the same time. For several years, the cellular industry has used the term to refer specifically to the Qualcomm-developed digital cellular telephone system based on CDMA techniques. Now that the technology is a standard, the preferred term is TIA IS-95 (Telecommunications Industry Association Interim Standard 95). In this paper I will tend to use "CDMA" to refer to the general characteristics of spread spectrum multiple access systems and "IS-95" to refer to the specific Qualcomm-developed CDMA digital cellular system.

CDMA System Advantages

CDMA's advantages over conventional narrow band systems include the following:

1. Automatic Transmitter Power Control

IS-95 adjusts mobile transmitter power with a fast "closed loop" power control system. It uses only the power actually needed at any instant (+/-1 dB) to produce an acceptable bit error rate at the cell receiver. This compares with conventional systems that need substantial transmitter power margins (often well over 20 dB) to "ride through" fade nulls. The much lower average transmitter power required by IS-95 results

in considerably less interference to other users sharing the same channel, and correspondingly greater system capacity and mobile battery life.

2. Forward Error Correction

IS-95 uses very strong forward error correction (FEC), decreasing required transmitter power still further. The typical E_b/N_0 (energy per bit to noise spectral density ratio) required for good performance is 6 dB or less, while FM typically requires a signal-to-noise of at least 17 dB.

3. Universal channel reuse

Narrow band systems cannot reuse every channel in every cell. Typically only 1/7 of the channels allocated to a carrier are usable in any given cell due to the need to protect neighboring cells. The inherent interference resistance of spread spectrum, however, allows every IS-95 cell to use every channel.

4. Variable rate speech coding

Narrow band cellular systems statically allocate transmission resources to each user for the duration of a call. Proposals exist for the dynamic reallocation of channels among users according to voice activity, but they are hampered by the inherent overhead. The problem is fundamental. It is similar to that of contention-based packet radio networks with short packets and long feedback delays.

CDMA, on the other hand, assigns each user a unique code (spreading sequence) from a set that is for all practical purposes infinite. There is no limited pool of single-user frequency channels or time slots to carefully manage. When a CDMA transmitter has data to send, it simply sends; when it no longer has traffic, it stops (or slows to an "idle" speed, as in IS-95). To be sure, the shared wide band channel still has a total capacity limit, but it is only the sum of many users' instantaneous demands that matters. The law of large numbers makes this sum much more predictable and uniform than the individual demands of each user. This allows each wide band spread spectrum channel to operate much closer to its capacity limit than if it was rigidly subdivided into fixed narrow band channels.

The IS-95 system uses a variable rate voice coder (vocoder). When the user speaks, the vocoder runs at a relatively high rate; when the user stops talking, the vocoder idles at a low rate. Every 20 ms, the vocoder produces one of four frame sizes: 16, 40, 80 or 171 bits, thus continuously adapting to the speaker's speech patterns. In normal conversation, the average data rate required by the variable rate vocoder is only about 40% of the peak rate.

To keep the transmitted energy per bit constant, IS-95 transmits with power proportional to the data rate. For example, it sends 1/8 rate data frames with 9 dB less power than full rate frames. This reduces average transmitter power, reducing the interference to the other users, increasing system capacity and prolonging battery life while allowing the closed-loop power control system to continue functioning.

5. Soft Handoff

IS-95 supports a novel "soft handoff" feature that sets up individual calls through several cells simultaneously. This allows the mobile to combine the independently fading signal components from several cells, and the base station to select the best cell to receive the mobile on a frame by frame basis. The system takes down a path through an old cell only when the mobile becomes firmly established in a new cell. As long as the two cells' service areas overlap, this "make before break" feature makes soft handoffs completely undetectable to the user. It also allows the use of less RF power for the same grade of service.

A single IS-95 RF channel is 1.25 MHz, the spreading bandwidth of the system. Each 1.25 MHz RF channel in a cell supports up to 61 traffic channels (simultaneous calls), but mutual interference limits the aggregate user data rate. That is, one can support as many as 61 simultaneous calls per channel per sector, as long as the users do not simultaneously transmit full rate frames. In practice, the independence of the speakers plus the law of large numbers reduce the probability of mutual interference under normal operating conditions.

IS-95 is similar to the TASI (Time Assigned Speech Interpolation) scheme that increases the carrying capacity of expensive undersea cables beyond that possible by dedicating transmission resources to each user. Both schemes are related to statistical multiplexing, the basis of packet data switching.

Adapting IS-95 for Data Services

Although Qualcomm originally designed IS-95 specifically for voice, its high performance digital radio modem and its use of variable data rates make it well suited for either circuit or packet-switched data. I have already demonstrated TCP/IP over IS-95, using the IS-95 radio channel as an access link to a router connected to the base station ahead of the vocoder. A radio link protocol (RLP) layered on top of the existing IS-95 architecture carries general purpose variable-length packets over the IS-95 radio channel. These packets can contain Internet datagrams or packets belonging to any other desired protocol suite, e.g., OSI CLNP or Appletalk.

This approach has two major advantages that should minimize cost and speed deployment. First, it minimizes the changes required to the existing IS-95 system architecture. The data service uses IS-95 only as a simple access link to a packet switch; the data packets themselves carry all network addressing information. Second, it maximizes use of existing and proposed Internet facilities, including those of special interest to mobile users such as the Mobile IP and IP Security protocols.

A Radio Link Protocol (RLP) For IS-95

In any mobile radio system, quality and system capacity are direct design tradeoffs and IS-95 is no exception. IS-95 provides an average frame erasure (loss) rate of 1-2% under credible full-load conditions. This figure maximizes system capacity without excessive degradation of voice quality with the chosen vocoder. However, when one considers that a 1 kilobyte data packet is about 50 full-rate IS-95 frames, and that at a frame erasure rate of 2% the probability of getting 50 consecutive frames across the link without any erasures is only about 36%, relying totally on end-to-end error recovery from protocols like TCP is clearly not a good idea. We need a little help from the link layer. This is the main job of the proposed Radio Link Protocol (RLP).

The author is still a strong believer in the End-to-End Principle, and did not design RLP for absolute reliability. In particular, it bears no resemblance to LAPB, the over-designed and inefficient link layer from X.25 that keeps popping up in "new" protocols, with little or no thought given to its appropriateness for the task at hand.

RLP is a "lightweight" protocol intended only to improve performance by complementing existing end-to-end error recovery mechanisms in protocols like TCP. RLP can still lose packets from several causes, but they are rare enough that relying on TCP to recover from them is appropriate; handling them at the link layer is well beyond the point of diminishing returns. Among the possible causes of packet loss are: dropped IS-95 calls (usually caused by the user moving outside the coverage area); errors undetected by the relatively weak frame CRCs already in IS-95 (12 bits for full rate, 8 bits for half rate); data errors within the IS-95 base station; and excessive requests for retransmission of a particular erased IS-95 data frame.

RLP uses negative acknowledgments (NAKs) to request selective retransmission of individual IS-95 frames. Since IS-95 frames get through much more often than not, this makes much more efficient use of the back channel than the traditional practice of sending a positive acknowledgment (ACK) for every successfully received frame. (Recall that IS-95 sends lower rate frames such as idles at a lower power level.)

RLP first encodes user packets in a PPP subset. As in conventional PPP, RLP adds a 1-byte Protocol ID to the front of the packet, adds a 16-bit CRC and a flag to the end, and queues the resulting byte stream for transmission. Byte stuffing maintains transparency in the user data wherever it contains values special to the framing technique (such as the flag, 0x7e).

RLP then divides the byte stream produced by PPP into IS-95 frames for transmission. Each IS-95 frame that carries user data is sequence numbered modulo 256. RLP increments the sequence number by one after each data frame, regardless of size; at full speed, the sequence numbers wrap around in 5.12 seconds. The receiver monitors incoming sequence numbers, and as long as none are missing, quietly passes the data portion of each frame up to a PPP receiver routine for reassembly into complete packets. Missing sequence numbers trigger NAKs from the receiver, requesting the sender to retransmit the requested frame(s). NAKs are retransmitted up to three times before the receiver gives up. RLP puts data frames with sequence numbers beyond those needed next to complete a packet on a reassembly queue until they can be used.

Idle frames also carry sequence numbers, but RLP does not increment them; their purpose is to "flush out" a NAK from the receiver when the channel erases the last data frame before an idle period. Because RLP repeats the same sequence number as long as the channel remains idle, even a long series of erased idle frames will eventually trigger a NAK when the first unerased idle gets through.

IS-95's synchronous nature makes this pure-NAK scheme work. As long as the channel remains up, it needs at least a 2-byte frame to transmit every 20 milliseconds, and this frame might as well carry the current sequence number. In contrast, an asynchronous channel (one that goes completely idle between packets) needs a positive acknowledgment to guard against loss of the last data frame sent before the link goes idle.

The current RLP uses half- or full-rate IS-95 data frames to carry user data, and eighth- and quarter-rate frames for idle and control respectively. Future versions may expand the available rates for performance tuning purposes.

Because the basic IS-95 system is connection-oriented while the packet service provided by RLP is connectionless, RLP automatically brings up and drops IS-95 traffic channels as needed. X.25 and dialup circuits have long used this same technique, except that IS-95 can bring up a traffic channel in only a second or so. RLP drops channels after an idle timeout, but the exact duration of the timeout is not too critical because an idle channel consumes relatively little of the IS-95 channel capacity. Contention for physical hardware resources (the modem ICs at the cell sites) is more likely than channel capacity considerations to be the controlling factor in deciding when to close idle traffic channels.

Prototyping and Testing

The prototype IS-95 data service uses PC clones at both the mobile and base stations running the KA9Q NOS TCP/IP package, with RLP and IS-95 interface drivers added. Since NOS includes an IP router, it was straightforward to use the prototype to access the regular Internet from a laptop computer plugged into a IS-95 mobile. The mobile laptop also supports additional computers via SLIP or Ethernet connections, although all of the remote computers must share the bandwidth of a single IS-95 traffic channel.

In a public demonstration in February 1993, I constructed two mobile data stations, each with a PC (running NOS TCP/IP/RLP) and a Macintosh Powerbook (running the University of Melbourne's Appletalk Remote Network Service over UDP/IP/SLIP). The demonstration supported simultaneous usage of the PC and Mac. The latter mounted Appleshare disk volumes on Qualcomm's file servers across the IS-95 radio link provided by the PC.

Field tests demonstrate the RLP's effectiveness. In tests from a moving vehicle in the San Diego CDMA Validation System, the packet loss rate was typically one in 10,000. Throughput on file transfers (using FTP/TCP/IP over the RLP) is typically about 900-950 bytes/sec, depending on the IS-95 frame erasure rate. TCP segment size is also a factor, although Van Jacobson header compression reduces header overhead significantly. The RLP seems quite robust; it hardly loses any packets unless the user moves outside the cellular coverage area, in which case there is little for it to do until the physical layer is restored. There is reason to believe that the dominant component in the 1:10,000 packet loss rate mentioned above was bit slips in the back haul links between the cell sites and the central switch, but the rate was already low enough to not warrant serious attention.

Future Activities

We have submitted the packet data architecture and radio link protocol to the TIA TR45.5.1.5 working group on IS-95 data services, where they are currently under consideration. Among the immediate needs of the working group are ways to provide conventional data services such as access to asynchronous modems on the regular public telephone network and support for conventional Group 3 facsimile, both of which have been prototyped and demonstrated over TCP/IP and RLP/IS-95 with the same software package developed to provide packet mode services. In this way, IS-95 has shown itself capable of supporting traditional data services while at the same time positioning itself to support the packet-based services that will certainly dominate mobile computing in the future.

Obtaining Specs

The complete Qualcomm CDMA system specifications (plus a system overview) are available by anonymous FTP from ftp.qualcomm.com in directory /pub/cdma. The files are in gzipped Postscript format.

An Infrared Network for Mobile Computers

Norman Adams
Palo Alto Research Center
Xerox Corporation
norman@parc.xerox.com

Rich Gold
Palo Alto Research Center
Xerox Corporation
richgold@parc.xerox.com

*Bill N. Schilit**
Computer Science Department
Columbia University
schilit@parc.xerox.com

Michael M. Tso
EECS Department
MIT
tso@lcs.mit.edu

Roy Want
Palo Alto Research Center
Xerox Corporation
want@parc.xerox.com

ABSTRACT

The ParcTab infrared network provides a flexible infrastructure for research into wireless mobile computing. The network consists of a collection of room-sized cells each wired with a base station transceiver. Mobile computers communicate with transceivers through a carrier sense multiple access (CSMA) protocol and act as terminals for applications executing on remote hosts. Each mobile computer is represented by a proxy, or *agent*, accessible to applications at a fixed network address. In the ParcTab system it is the agent that is responsible for delivering requests to its corresponding mobile computer, and tracking the mobile as it moves from cell to cell.

1 Introduction

Personal digital assistants (PDAs) perform the function of paper organizers with the added benefits of digital control. Like paper organizers, PDAs fit in the palm of the hand and can be carried with you throughout the day. However, the true advantage of PDAs are realized by communication with networks of more powerful computers, enhancing the capabilities of these miniature machines.

We have built ParcTab and a supporting infrastructure. ParcTab is a PDA that communicates using infrared (IR) data-packets to a network of IR transceivers. The infrared network is designed for in-building use, where each room becomes a communication cell. In contrast to the approach used by other PDAs, most ParcTab applications run on remote hosts and therefore depend on reliable communication through the IR network. The infrastructure provides reliability as well as uninterrupted service when a ParcTab moves from cell to cell.

This paper focuses on the Unix-based infrared network we have built in support of the ParcTab. The first section provides a system summary. The next three sections describe the physical, data link, and network layers. We conclude with a performance assessment.

*Currently visiting Xerox Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA 94304

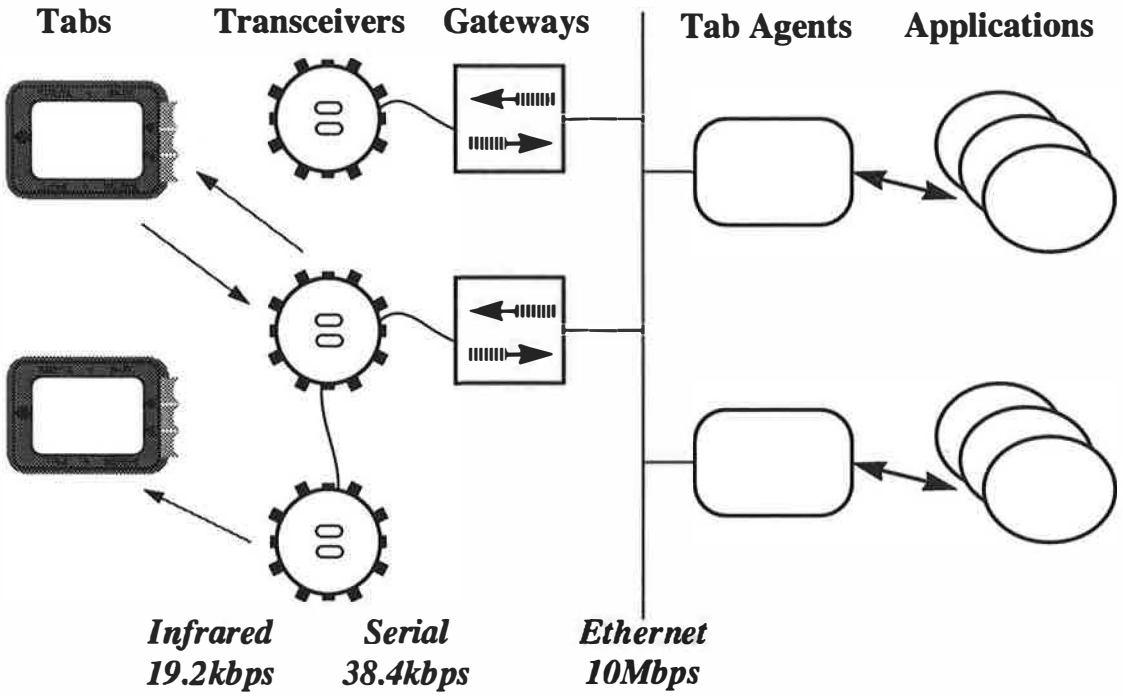


Figure 1: PARCTAB System Components

2 System Summary

The PARCTAB IR network provides communication between stationary transceivers (*base stations*) and mobile systems[1]. Although the IR network was designed specifically for a hand held PDA called the PARCTAB (hence the name of the network), the network is used with other portable computing devices. This section presents an overview of the PARCTAB system components.

The system components are shown in Figure 1. Tabs and transceivers are custom hardware components. Gateways, agents and applications are Unix processes. The system employs three communication media: infrared is used for packets between transceivers and tabs; a serial line connects a host running an IR gateway and a base station transceiver; and an Ethernet connects IR gateway processes with agents. Sun RPC is used for communication between gateways, agents and applications. These components are described below.

The PARCTAB PDA is designed to fit in the palm of the hand and incorporates three buttons as part of the grip. Output is displayed on a touch sensitive 128 x 64 pixel resolution display. A number of IR diodes for transmission are spaced around the tab case, and a multi-directional receiver is built into the top of the unit. The tab functions more as a graphics terminal than a general purpose computer. However, the tab interface includes functions to download object code and data.

One or more base stations can be connected to a host workstation through a serial port. A device control protocol is used between the host and its connected base stations. In the case of multiple base stations sharing the serial line, the control protocol is synchronous: each base station must be polled by the host before it can send data. This configuration is useful for long runs with low expected utilization (e.g., hallways). For better efficiency, the host serial line can be dedicated

[1] Mobile to mobile communication is part of the design but has not yet been implemented.

to a single base station, and the base station operates asynchronously: the base station forwards IR packets to the host as soon as they are received.

Gateway processes on each host export a Sun RPC [2] interface for access to the base station. The RPC interface allows clients to send a packet unreliably over the infrared medium. When transceivers receive packets from the gateway they broadcast them into the infrared medium. The packets sent from the transceiver contain type, length, source address, destination address, payload, and checksum. Above this protocol layer is an at-most-once RPC employing sequence numbers and timeouts.

Our experimental network uses baseband modulated infrared to carry variable length data packets with a maximum size of 256 bytes. A packet broadcast into the infrared band is received by all transceivers within a room-sized "cell," and is copied from the infrared medium by destinations which select it according to the packet's address. To handle demands for growth, more transceivers can be installed on other workstations. Infrared transducers have the benefit that they are small, low power and low cost. Since infrared cannot penetrate opaque materials such as walls and doors, the network has good isolation properties.

This system provides a flexible infrastructure for research into mobile computing. We're currently designing other mobile devices that will use the same gateway and transceiver network. Over a period of time, the system can be scaled to support many mobile computers, as the incremental cost of extending the network is small.

3 Network Transceiver Hardware

The IR transceiver contains electronics to send and receive signals in the IR medium, verify checksums, and buffer transmit and receive packets in a FIFO. The transceiver is also capable of sending and receiving packets at multiple data rates and generating data-link layer acknowledgment packets. The current version of the transceiver is capable of transmission over the infrared medium at 9.6kbps and 19.2kbps. Although this is slower than many radio-based networks, the small cell size permits a high aggregate bandwidth for multi-cell systems.

We have designed IR transceivers in four different packages. Both the PARCTAB and a pad-sized mobile computer contain built-in transceivers. The base station is an externally powered transceiver along with a serial-line interface. This unit fits into a 4-inch square clear styrene box, and can be attached to a ceiling, a bookshelf, or simply placed on a desk in rooms wired as cells. Finally, we are building a self-powered transceiver, the *velcro-station*, for use with a wider range of portable machines such as the HP-95 and Apple Powerbook.

The infrared communication capabilities of the PARCTAB network are somewhat different from a wire-based Ethernet [6]. Quantitatively, our IR network transmits at 19.2kbps, or about 500 times slower than a 10Mbps Ethernet. The infrared network is also more susceptible to environmental noise caused by, among other things, bright sunlight and fluorescent lights. Qualitative differences include the fact that packets might traverse multiple paths from the mobile host because of overlapping transceiver cells. Finally, the infrared transceiver is able to listen for a clear channel before sending, but because of hardware limitations, it is not possible to detect collisions once transmission has started.

The rest of this section describes transceiver functions. We first explain how the base station transceiver communicates using device commands over an RS-232 serial line. This is followed by a description of the infrared physical layer and media access layers which are implemented by a microprocessor within the transceiver.

3.1 Transceiver Device Control

The base station and self-powered velcro-station connect to a host through a serial-line interface. There are a number of reasons for this design instead of, for example, a bus interface. First, the serial-line transceiver can be used with a number of machines containing different bus architectures

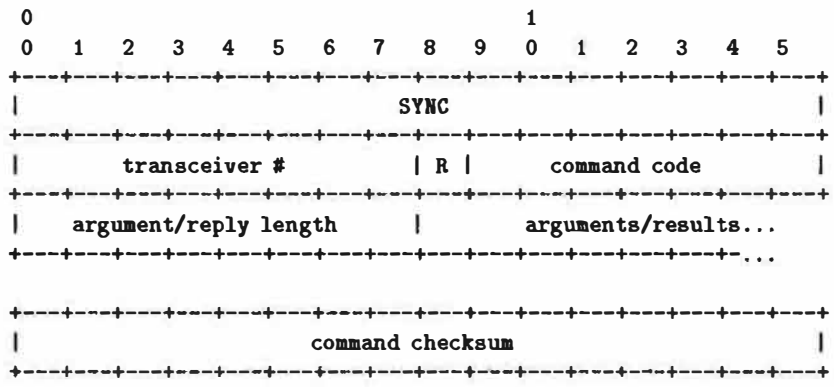


Figure 2: IR Transceiver Command Format

without modification. Indeed, some mobile systems do not have extension bus slots whereas they almost always have serial-line ports. Serial-line devices are low cost and easy to install. In the case of a base station they can also be configured in daisy chains. Finally, although serial interfaces are slower than DMA interfaces, this has not affected our current system because the IR interface is also quite slow.

Communication between the base station transceiver and host workstation is achieved by a command/response control protocol. Commands are sent to one of the transceivers on the serial line and the transceiver responds with a command result. The four main purposes of the control protocol are: to allow for multiple transceivers, to provide framing for outgoing IR packets, to detect errors, and to configure the operation of the transceiver.

The command format is shown in Figure 2. The first two bytes are a synchronization signal that uniquely identify the start of a command. Any `SYNC` bytes contained in the data are quoted. Following this is the transceiver number which selects a transceiver on a serial line (for multiple-transceiver configurations), the command code, arguments, and checksum. The checksum is necessary because we do not assume error-free communication over the serial line to the transceiver. The `R` field of the command code is set for all command responses from the transceiver.

The command format provides a well defined interface for transceiver-host interactions. The command set implemented by transceivers has evolved over time and consists of functions for debugging, configuration, and transmission. Debugging commands include setting a loopback mode and a mode that ignores checksums. Configuration commands permit device and FIFO reset, setting an address filter, and setting asynchronous mode so that incoming packets are immediately passed to the host. Finally, data transmission is accomplished with a command to broadcast the command argument over the IR medium. When asynchronous mode is disabled, infrared packets are read using a poll command.

3.2 Infrared Physical Layer

Infrared light emitting diodes (LEDs), similar to those used in the remote controls of consumer electronics, are the transducers used to implement the physical layer of the PARCTAB protocol. These diodes emit light at a wavelength of 850nm. In order to guarantee the IR emissions are diffuse and hence they will travel in all directions in a room, wide-angle LEDs are used. For reception, detector diodes in conjunction with a preamplifier demodulate the incoming signals.

Information is encoded onto the IR carrier by switching the carrier on and off in such a way that narrow pulses of IR radiation are emitted and the gap between the pulses carry the channel information. The modulation technique is called pulse position modulation. The emitted pulses are the delimiters for the encoded data and these have a constant duration of 4us. This signal encoding

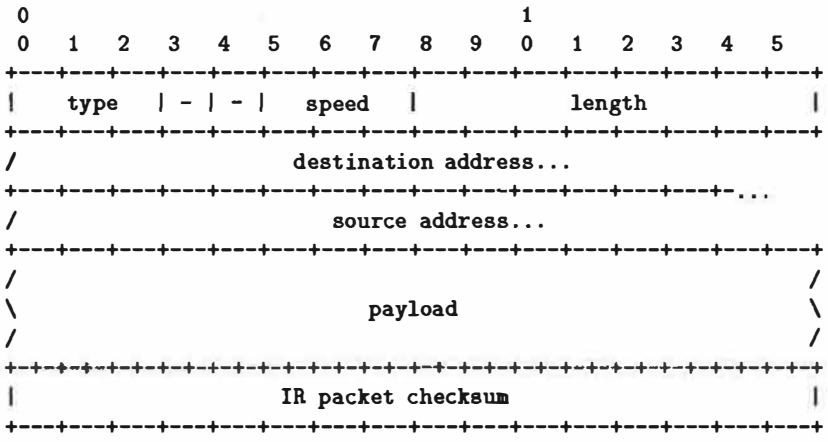


Figure 3: Link layer IR Packet Format

has the property that the average energy emitted is small. If a greater intensity of radiation is required to ensure a suitable signal range, the pulses can be emitted at a greater intensity and hence a greater instantaneous power. However, to compensate the average power can be kept constant by decreasing the duration of the transmitted pulses.

3.3 Infrared Media Access Layer

The protocols used to access the IR medium in the ParcTab system are experimental and will continue to be developed and improved in future work. At present, the system uses a carrier sense multiple access (CSMA) protocol without collision detection or collision avoidance. Lost packets are handled by mechanisms implemented in the higher protocols layers. However, the protocol design allows for data link layer acknowledgment of packets through the use of an option in the packet header. The acknowledgment from the destination transceiver has a good chance of being received without collision because it is sent immediately upon packet receipt.

ParcTabs are used as terminals and therefore the average IR packet size for downlink communication tends to be much larger than the uplink packet size. Typically, uplink packets—from mobiles to room transceivers—carry key presses and pen events which are small. Downlinked packets carry display updates, such as bitmaps and text, and are comparatively large. All downlink packets sent to a communication cell are serialized by the cell's base station and therefore do not collide with each other. Packet collisions can be caused by the asynchronous button presses and pen events from the tab, or by base-stations in overlapping cells.

The communications cells used in this system are very small and are sometimes referred to as "nanocells." In other wireless mobile networks, variants of the CSMA/CA protocol, e.g., MACA [5], have been suggested to improve bandwidth utilization in the event of "hidden terminals". It is our belief that hidden terminals are not a significant problem between cells in a cellular system where the cells are well isolated. Also, in the ParcTab network, hidden terminals within one cell do not tend to occur because the cells are small and the emitted power from one mobile can nearly always be sensed by another.

3.4 Link Layer Packet Format

The tab network packet format is shown in Figure 3. Our design goal was to provide sufficient flexibility so that encapsulating other protocols would be easy and efficient while providing the low level capabilities which we believe are important for mobile protocols. The **type** field is used to identify packet formats, allowing multiple IR protocols to coexist with this system. The **speed** field allows sending at multiple baud rates, currently either 9.6kbps and 19.2kbps baud. The type byte

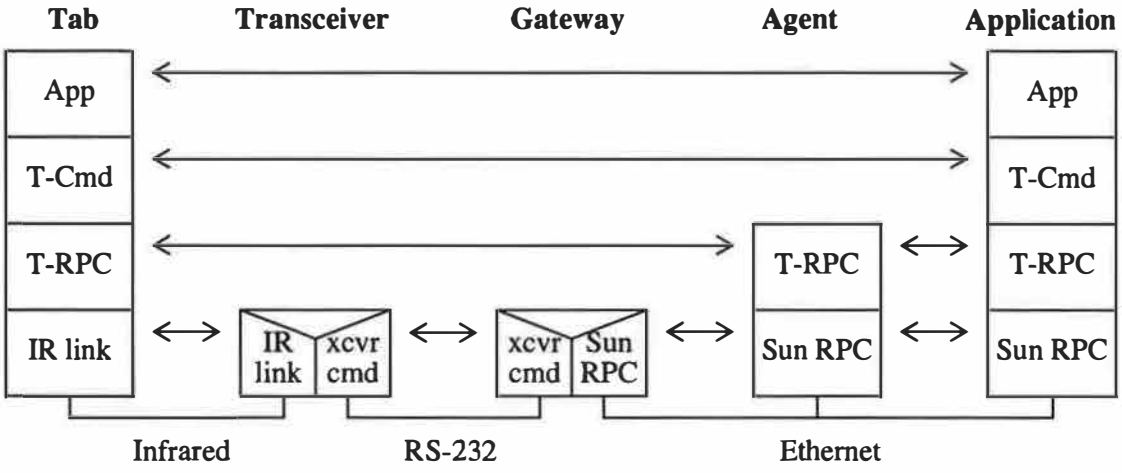


Figure 4: PARCTAB Network Layers

is always sent at the lower speed. The checksum is computed using an algorithm [3] that is suited to the limited arithmetic capabilities of small microprocessors.

4 Network Layer Implementation

The functions of the network layer include routing and heterogeneous network operation. In broadcast networks, such as Ethernet, the routing of packets is quite simple. In the case of a mobile host network, routing can be more complicated.

The network layer is implemented by a collection of independent processes called *tab agents*. Each tab is represented by an agent which is the only component that must know how to route packets to the tab. The tab agent serves many of the same functions as Mobile Support Routers (MSRs) in the Columbia Mobile IP proposal [4]. However, instead of employing MSRs to handle routing for a group of cells, our network assigns agents to handle routing for specific mobile hosts.

IR packets are received by one or more transceivers and transferred to the cell's IR gateway. A packet's source address is used to forward the packet to the tab agent. In order to translate between an infrared tab address and an agent process's Internet address, a name service is used. IR gateways use an address cache to avoid referencing the name service on each packet.

When packets are sent in the opposite direction, from application to the tab, the location of the mobile host is required. Location information is maintained at the agent by the use of return addresses and beacons. When an IR packet arrives at the agent it contains a return address for the cell that generated the packet. Any packet destined for the tab is sent to the most recently received cell address. The tab agent caches these addresses. This approach is adequate as long as tabs generate packets when moving from one cell to another. Because users cannot be relied on to create events to update their location, the tab "beacons." The beacon is a tab-generated event that is sent periodically whenever the tab is idle. Like all tab events, the beacon includes a return address which the agent uses to update the cell address for the tab.

5 Application-Tab Communication

This section describes communication between the tab and applications running as Unix processes on remote workstations (see figure 4). There are two styles of communication in our system: asynchronous (one way) events from the tab, which are unreliable; and at-most-once RPCs between the applications and the tab, which are reliable. These are described below and illustrated by an

example.

5.1 Application-To-Tab RPC

ParcTab applications use a command protocol (T-Cmd), see figure 4, for controlling tabs. Each T-Cmd operation corresponds to a particular tab capability such as displaying bitmaps or generating a tone. T-Cmds are converted to a packet format suitable for transmission to a tab using T-RPC. A round trip T-RPC involves Ethernet, serial line, and IR net transmissions, and provides reliable packet delivery to a mobile host. T-RPC addresses two problems: lost packets, caused by collisions and transmission errors; and movement of tabs between cells.

The T-RPC is similar to the approach described in [1]. A T-RPC request is sent to the tab (actually broadcast from the cell transceiver where the tab was last sighted). If no response appears within a certain delay, then the packet is resent. If a packet is seen by another cell's transceiver, then the T-RPC request is immediately resent to the new location. Sequence numbers are used to implement an at-most-once RPC mechanism. Tabs receiving a duplicate request respond with the previous reply. In the case that the tab receives a T-RPC request while it is still executing the previous request, it will send a reply indicating it is busy. The T-RPC implementation backs-off the retransmission of requests when the tab is missing or busy.

Reliability at the T-RPC level instead of lower levels of the tab protocol has simplified our system design. Our original design proposed that the gateway (MAC layer) would retransmit packets for some period, and if that failed, then the tab agent (network layer) would retransmit to a different gateway. Event though an extra Ethernet RPC is necessary in the retransmit loop when retransmission occurs at the agent, our current design is effective because Ethernet communication has much lower latency than tab infrared communication.

5.2 Tab-To-Application Events

The tab transmits button and pen events as IR packets. These events are not acknowledged by the receiver and are therefore unreliable. It is left to the user to retry a button or pen action until a response is seen.

The agent can receive duplicate events when multiple transceivers receive a tab event and each forwards a copy to the host. The agent filters duplicate events and discards old events rather than delivering them out of order. Sequence numbers are used for both of these purposes. An event is ignored unless its sequence number is greater than the sequence number of the previously processed event.

5.3 Example

One ParcTab application draws a piano keyboard and accepts pen events that cause tones to be played. The low-level communication involved in this process is described below.

- The application generates a bitmap draw request for the tab to execute. The application sends a T-RPC request to the tab agent and then waits until the agent responds with a reply from the tab before continuing.
- The agent receiving the application T-RPC sends a packet to the IR gateway where the tab was last sighted. Since the agent is responsible for reliable delivery, it will resend until a valid T-RPC response is received.
- The IR gateway receiving the packet encapsulates it in a transceiver command and transfers it over the serial line to the base station, which in turn broadcasts the packet over the infrared medium.
- When the ParcTab receives the T-RPC request, it draws the bitmap and generates a T-RPC response which it transmits over the infrared medium. Any base stations receiving the IR

packet transfer it over the serial line to the cell's IR gateway.

- The IR gateway upon receipt of an IR packet from the serial line strips off packets headers and forwards the packet to the tab agent.
- The tab agent receives the packet containing the T-RPC response and matches it to the pending T-RPC request. This causes the T-RPC to return to the application.
- The T-RPC containing the bitmap draw had the effect of displaying a button on the tab display. The user tapping the touch-sensitive screen generates a tab pen event that is broadcast over the IR medium. Transceivers receiving the event write it to the serial line for processing by the IR gateway. The IR gateway forwards the event to the tab agent, and the tab agent forwards the event to the application.
- When the application receives a pen event it sends a T-RPC packet containing a tone command making the tab play a note in response to the keyboard button being pressed.

6 Performance & Analysis

The distributed infrastructure supporting the ParcTab network is built from a collection of Sun Microsystems SPARC 2 workstations[2] running SunOS 4.1.3. and are connected by a 10Mbps Ethernet.

6.1 System Measurement Methodology

To monitor the behavior and performance of the system we use three sources of information. First, programs include a large number of trace statements. A trace statement is a print statement that executes only when its controlling trace variable is set. The programmer defines the set of trace variables for a program, and associates a trace variable with each trace statement. The trace variables in a program are collected together and can be viewed and modified during the program's execution. Each program also exports a network accessible command interpreter that the implementor can use to change parameters referenced by the program during execution. Second, we used a kernel events tracing package. By inserting `vtrace` system calls into programs, and by running the IR gateway, agent and application on the same machine, we are able to get a precise time line of events across all the Unix processes in this system. Third, we used an oscilloscope to determine the timing of relevant hardware events:

- serial-line transmission of T-RPC request from host to transceiver
- IR transmission of T-RPC request from transceiver
- IR transmission of T-RPC reply from tab
- serial-line transmission of T-RPC reply from transceiver to host

Using this information we removed a number of performance bottlenecks in the system. We found that Sun's serial-line driver imposes a 20ms delay between delivering consecutive buffers of data received on the serial line. This delay was removed by writing a loadable device driver. In addition, trace statements were expensive even when not selected to print, because the parameters were always evaluated. Some of these parameters invoked procedures for the textual formatting of large arrays of data, and hence the delay.

When the original measurements were performed, each tab packet contained only a single tab function. We have since implemented the grouping of multiple tab functions into a single command packet[3], thus increasing the average size of packets and improving overall throughput.

[2] A Sparc 2 workstation typically has an instruction rate of 20 MIPS.

[3] Grouping works for functions that do not return a value other than a success or failure. The tab will process a list of functions contained in a command buffer in order until all of them have been executed or an error occurs.

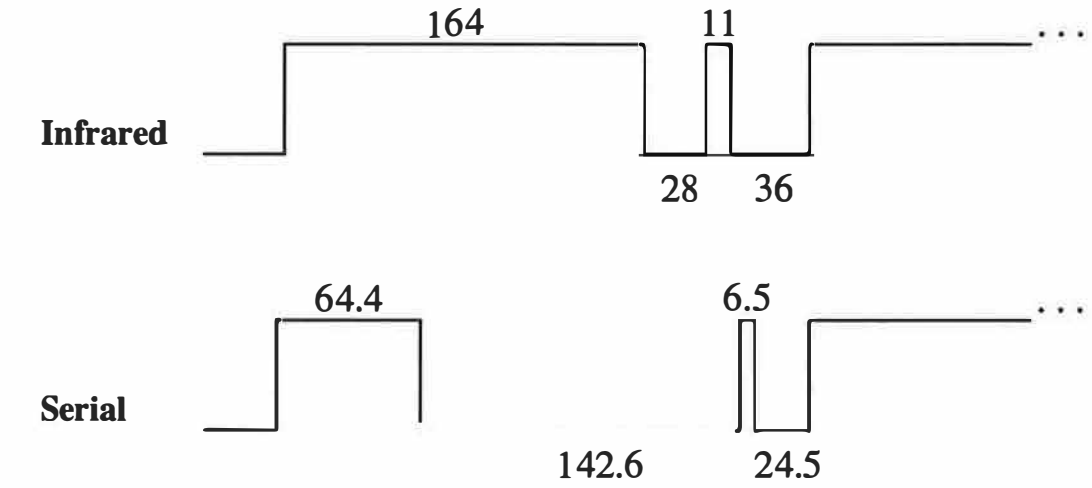


Figure 5: Infrared and Serial Packet Transmission Times (ms)

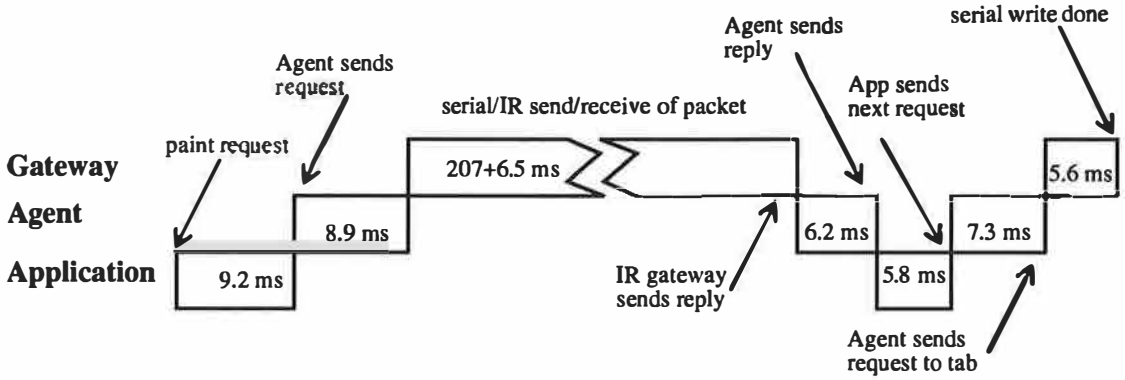


Figure 6: Unix Process Event Trace

6.2 System Timing

Figure 5 charts the infrared and serial-line activity in the course of transmitting a 240-byte packet to the tab. The sequence starts with a 64.4ms transmission on the serial line of the packet plus 7 bytes of transceiver command overhead. The transceiver begins broadcasting bytes over the IR medium 4ms after it begins receiving the packet. The IR transmission time is 164ms since the infrared data rate is 19.2kbps versus 38.4kbps for the serial line. The 28ms idle period following the IR transmit is taken up with tab execution time for the T-RPC request contained in the packet. Following this event, the IR transmission of the T-RPC reply, a 14-byte packet, takes 11ms. The transceiver transfers the 14-byte packet plus 7 bytes of transceiver command overhead on the serial line in 6.5ms. There is no overlap between serial transmission and IR transmission in this case because the packet checksum is verified before forwarding it over the serial link. The remaining 24.5ms before the next transmission is due to communication between the IR gateway, agent, and the application as shown in Figure 6.

7 Conclusions

The general approach used by the ParcTab system has proven to be beneficial from many standpoints. The time and effort required to install a communication cell in a building where networked workstations are already in place is small. Typically we find an IR transceiver can be installed in about 15 minutes. By making use of existing workstation serial ports, the network-management costs are also low. Moreover, because the project could enforce its own policies at a level higher than the Unix system infrastructure, it was not necessary to involve the system administrators in setting up this experimental system.

Another success of the project was in the choice of general protocols for communication across the serial line and through the IR medium. The serial-line protocol allowed us to transfer IR packets from a workstation to a transceiver. Once in place this code remained static through most of the ParcTab system development. The same is true for the IR-packet format. The data payload contained in these packets did however change to suit the functionality of the tab as we experimented with different UIs and data compression techniques. The interfaces chosen for separating the transceiver and the IR packet delivery system have stood up to many architectural changes without modification. In fact, the transceivers could deliver the packet structure of other protocols such as IP or ATM without change. We conclude the interfaces for these protocols were well chosen for this research vehicle.

In the initial design phase of the system there was concern that building an architecture to support the ParcTab distributed among many Unix processes, on two or more machines, would lead to unacceptable latencies when implementing interactive mobile-applications. In practice, for our design this has not been the case. Most round trip delays are under 250ms, hence comparable with human reaction times, and the system feels responsive in its use. This is particularly important so that the ParcTab gives an impression the computing power is palm of the user's hand.

A competitive design for the ParcTab network would be to implement Mobile IP [4]. This approach conflicts with the goal of designing a small, portable device and a low-cost infrastructure to support it. Mobile IP would require a tab resident implementation of IP, and the transceivers would become IP routers. These system requirements are beyond the capabilities of both the current base station hardware and the ParcTab.

8 Future Work

There are still many improvements that would enhance the tab system. The flexibility designed into our own system has not been fully explored, for instance, enabling link layer acknowledgment for tab event-packets will change the characteristics of the system bringing improvement in reliability, but at the same time, a possible reduction in packet throughput. The media access protocol in use could be replaced by other protocols, such as MACA [5], designed for the mobile environment. Many of these questions are not easy to answer by simulation simply because the usage model is not known. The existing ParcTab system allows us to gain understanding of the usage model and to plan for the future.

Currently, many companies are on the verge of making communication products that could significantly improve the bandwidth provided by the IR transceivers used in the experimental tab system. A 1Mbps product suitable for use with laptops should be available by the end of this year. Upgrading the system's IR physical-layer will not require changes in the packet delivery mechanism provided by the rest of the architecture. Incremental changes like this ensure a flexible development path and the longevity of the system.

Mobile computing provides many new opportunities for us to develop novel applications. The most important new dimension these devices bring is **location** and hence context information. Location tells the system where you are and context tells it who you are with. The authors have already had some experience with another device, the active badge [7] which provides location information about people who wear it in a suitably wired building. There are many applications

that could modify their behavior based on location. Programs can be designed to automatically make decisions governed by context rules that have been specified by a user. The investigation of this new environment is part of on-going work and will be the subject of future papers.

In summary, the PARCTAB system is novel and open for experimentation. It allows research into the field of mobile computing and also allows investigation of new applications in advance of commercial developments in this area.

References

[1] A.D. Birrel and B.J. Nelson. Implementing remote procedure calls. *ACM Trans. On Computer Systems*, 2:39–59, Feb 1984.

[2] John R. Corbin. *The art of distributed applications : programming techniques for remote procedure call.* Springer-Verlag, New York, 1991.

[3] John G. Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, COM-30:247–251, Jan 1982.

[4] J. Ioannidis, D. Duchamp, Jr. G. Q. Maguire, and S. Deering. Protocols for supporting mobile ip hosts. Technical Report RFC xxxx, Mobile Hosts Working Group, June 1992. *Draft RFC available by anonymous ftp from site parcftp.xerox.com:/pub/mobile-ip/columbia-draft-june-92.*

[5] Phil Karn. MACA - a new channel access method for packet radio. In *Proceedings, 9th Computer Networking Conference*, pages 134–141. ARRL/CRRL Amateur Radio, Sept 1990.

[6] Robert M. Metcalfe and David R. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7), Dec 1976.

[7] Roy Want, Andy Hopper, Veronica Falcao, and Jonathan Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 1992.

Unix for Nomads: Making Unix Support Mobile Computing

Micheal Bender, Alexander Davidson, Clark Dong, Steven Drach, Anthony Glenning, Karl Jacob, Jack Jia, James Kempf, Nachiappan Periakaruppan, Gale Snow, Becky Wong[1]
Nomadic Systems Group,
Sun Microsystems Computer Corp.
2550 Garcia Ave., Mail Stop MTV17-08
Mountain View, CA, 94043

Abstract

Traditionally, the Unix operating system[2] has been associated with deskbound machines tethered to the wall by a power cord and an Ethernet cable. Making Unix support a more nomadic model of computing requires changes in the entire system, from the kernel level through the user command set of applications. In this paper, we present the results of an experimental prototype development effort targeted at supporting a nomadic computing model in Sun's Solaris 2 SVR4-based platform[3]. The development involved enhancements in four primary areas: kernel changes to support power management and checkpointing of system state, drivers and other kernel support for the new PCMCIA bus standard, support for serial line networking, and a new electronic mail application designed specifically for accessing mail over slow serial connections. The paper discusses enhancements and modifications to the design of standard Solaris system components in each of these areas.

1. Introduction

An implicit assumption in the design of many Unix platform software components is that the workstation or server will remain tethered to a particular physical location by its need for power and an Ethernet connection. Newer, low power chip and system designs have enabled the manufacture of small form-factor, battery-powered portable computers. These computers are much more mobile than the machines on which Unix traditionally runs. Such machines may be used both in an office setting and while on the road, so that they become the primary machine for some users. A *nomadic computing model*, where users set up their computer and work somewhere then move the machine to a different physical location and continue working, is fundamentally different from the traditional deskbound model which Unix currently supports. Design modifications to a Unix platform for supporting nomadic computing require a system approach, since the nomadic usage model affects everything from the kernel through user applications.

The main factor driving kernel changes for nomadic use is power. Mobile computers tend to have tighter constraints on power consumption than desktop machines. These constraints derive primarily from the limited performance of existing batteries. Since battery technology is improving slowly relative to processor and memory technology, hardware and software for saving power in nomadic computers is essential to increasing the available compute time with limited battery life. In addition, nomadic usage patterns involve frequent power cycling, so that long start up and shut down times are less tolerable than in desktop machines. Mobile users want the ability to simply shut the machine off and later start it up again, having it restored to exactly the same state as they

1. Author's names are in alphabetical order.

2. Unix is a registered trademark of Unix System Laboratories.

3. In this paper, SunOS 5.x refers to SunSoft's implementation of Unix System V Release 4 (SVR4). Solaris 2.x refers to SunSoft's SunOS 5.x-based application development and delivery environment, which includes the OpenWindows window environment and the Tooltalk application integration environment.

left off so they can begin work immediately, because their work is frequently interrupted by movement. While similar functionality has been added to MS-DOS[4] and SunOS 4.1 [Tadpole93], to our knowledge, our work is the first attempt to add power management and checkpoint/resume to a version of SVR4.

Nomadic users may have limited access to the standard Ethernet connections which many desk-bound machines require. A wide variety of connectivity options, such as FAX/modem, ISDN, wired/wireless LAN/WAN, etc., may be available at the location where the user is working. In addition, the higher power requirements and large form factor of standard desktop bus extension peripherals are often inappropriate for mobile machines. A new bus standard for low power, small form factor (credit-card sized) hardware devices, called PCMCIA, has been designed specifically for portable computers [PCMCIA93]. Support in the Unix operating system for PCMCIA is needed to make PCMCIA devices available for Unix nomads. In addition, since many of the connectivity options open to nomadic users are likely be serial lines with relatively low bandwidth and high latency, transparent access to an efficient serial line protocol is required.

Perhaps the canonical application for mobile users is electronic mail. A variety of new connectivity technologies, including mobile cellular and packet radio networks, are enabling wide area access to electronic mail beyond the desktop. A critical factor in providing mobile users with electronic mail service is the communication protocols used between the client and servers. Currently, they are optimized for high bandwidth, low latency connections. Most of the newer, wide area connectivity options are low bandwidth, high latency, with a relatively high cost per packet. New application level protocols are required to deal with the different characteristics of these connections.

This paper presents an overview of changes in various Solaris 2.x system components necessary to support nomadic computing. We discuss adding support for power management, kernel and application checkpointing, PCMCIA, serial line connectivity, and mobile electronic mail. The next section describes the power management framework. Section 3. discusses the design of checkpointing, and how it interacts with power management to provide the user with "instant on" capability. In Section 4. , we discuss the implementation of PCMCIA on SunOS 5.x. Section 5. provides an overview of the serial connectivity and the asynchronous serial line link manager. In Section 6. , we discuss electronic mail as an example of what changes may be required in applications to support mobile users. Finally, Section 7. summarizes the paper.

2. The Power Management Framework

Limited power availability is the primary constraint on nomadic computer operation. Since a portable computer consists of a variety of hardware devices, conservation of power can be achieved by matching the power consumed to the activity level of each device. The objective of the power management framework is to match the power consumed by a device to its activity level. In its simplest form, the power management framework can turn devices off when they are idle and turn them on again when they are needed. If the device has special power management hardware, the power management framework can match the power consumption of the device to its activity level in a more fine-grained manner. The result is increased battery life, and therefore increased operating time.

The goals of the power management design are the following:

- Provide a generic, portable power management framework within the SVR4 system design,
- Operate transparently to the user, but include user tunable parameters,

In pursuit of these goals, the framework is divided into three basic parts:

- The device driver mechanisms to support power management,
- The pseudo device driver, `pm-driver`, providing supervisory kernel mechanisms and policies,
- The DDI/DKI [Sun92] changes required to support the framework.

4. MS-DOS is a trademark of Microsoft, Inc.

Figure 1 contains a diagram of the power management framework, showing the architectural relationship between the parts, and the parts themselves are described in a separate subsection. In the figure, a light grey arrow indicates that the function is optional, a solid arrow indicates it is required, and a striped arrow indicates the functionality is provided by an implementation dependent mechanism.

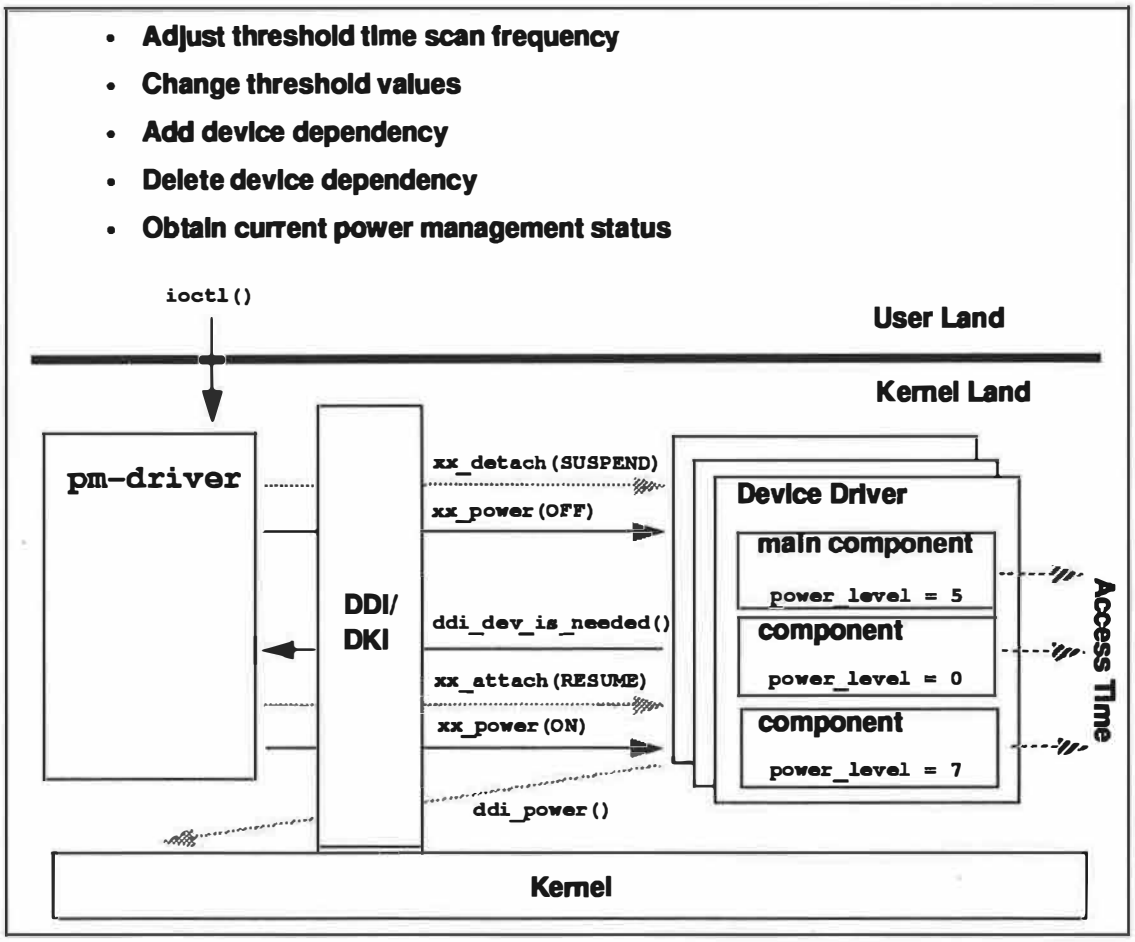


Figure 1 Power Management Framework

2.1 Device Driver Mechanisms

Device drivers are really the key to power management. Device drivers control when a device is physically accessed, whether a device can be suspended, and whether any special power management hardware components are available to the kernel. The power management framework depends on the device drivers to manage the individual devices they control, which is the bulk of the power management. To remain compatible with existing drivers, the framework assumes that it should not try to power manage a device if the driver provides no power management.

The design of the power management framework required adding a new state to device drivers, and rethinking how device driver software relates to the hardware it controls. The new state is device *suspension*. Suspension involves bringing the device to a state where there is no activity, although the power is still on. From a software point of view, suspending the device requires saving hardware registers and driver state to memory. Device suspension is clearly separated from power off, and new driver entry points were added for both suspension and power off. Powering a device off usually involves a writing specific value to a hardware register or issuing a device-specific command to the device.

From the power management point of view, hardware devices can consist of a variety of parts having differing power management capabilities. The extreme example is the entire computer itself, which consists of disk, CPU, display, etc. The power management frameworks models this structure by specifying that a device consists of a number of power manageable hardware units, called *components*. Each component is separately power controllable and has its own set of power parameters. One component is defined to be the main component and is logically where the state of the device is kept. The main component requires the driver to be suspended to save the hardware state before it can be turned off. The driver must also be resumed to restore state when the main component is turned on. The other components must be stateless and do not require the driver to be suspended or resumed to be power managed.

Components collect together state and functionality relevant for power managing a device. Each component has a power level associated with it. A zero power level indicates that the device is turned off, while a device at a nonzero power level is on. The component is responsible for mapping between power states and integer power levels provided by clients to control power. Each component has two basic functions:

- It must export its last access time. A component indicates that it is busy by exporting a zero access time, allowing the framework to detect when the component does not need power management.
- It must notify the power management framework before it is used, if power has been reduced below the level needed for the operation. The component need not notify the framework if the power level is sufficient for the operation.

In some cases, power management within a driver could conflict with efficient device access. The granularity of power control is entirely up to the device driver writer. Each component of a device may be tuned differently, depending on where in the driver the timestamps are updated. Tuning of a component can assure that critical paths within drivers avoid any overhead at the possible expense of some power management granularity. The framework allows the driver writer to make the necessary trade-offs between device access efficiency and power manageability.

2.2 Power Management Pseudo Driver

The power management pseudo driver `pm-driver` is responsible for managing power reduction in the system. The `pm-driver` periodically scans all devices, querying each power manageable device about its usage, and informing the device to shut off if the device has been idle longer than a set amount of time. Additionally, a power manager interface is provided through `ioctl()` operations within the driver. The `ioctl()` operations allow the user to tune the parameters of the `pm-driver`'s power management policies. The `pm-driver` does not power manage any devices not exporting the correct power management information. The `pm-driver` initiates powering down of a component due to inactivity, and powering up of a component through a request from the device driver. If a driver is in a state such that it cannot save all the necessary information, it may return failure. In that case, the `pm-driver` does not turn the main component off immediately, but it may try again later.

Through `pm-driver`, the power management framework recognizes two types of dependencies between devices: logical and physical. A power manageable device has a physical dependency if it has one or more devices attached below it on a set of interconnected buses. A device has a logical dependency if no physical interconnection exists, but a power management constraint exists nevertheless. For example, a graphics card driver that relies on keyboard and mouse inactivity to determine if the display can be turned off has a logical dependency. Prior to power removal, all dependent devices must be properly suspended and powered down before the dependee can be shut down. Physical dependency between devices is implicitly defined in the kernel device tree. The `pm-driver` must be informed of logical dependencies either through a driver configuration file or through an `ioctl()` call. If either a physical or logical dependency exists, then no component of the device is powered down unless the dependency is satisfied.

In order to decide if a device component is idle, the `pm-driver` obtains the time that the component was last accessed, which has been exported by the component. If the last access time is zero, the

device is busy. If the difference between the last access time and the current time exceeds the management policy threshold time for the component, the component is deemed idle. If all logical dependents of the device are also idle, the `pm-driver` attempts to turn off the device. If an error occurs, the power down fails and the component is left on.

The `pm-driver` provides all kernel mechanisms and implements power management policy. It is accessed by the user through `ioctl()` operations. These operations allow the user to adjust the frequency of device scans, change component threshold values, or add and delete dependencies, and to obtain the current power management status of a device or of the `pm-driver` itself. The `pm-driver` only controls devices that have been configured (i.e. have at least their threshold time(s) set). Typically, at boot time, a script runs a configuration program that reads a system file and configures all power manageable devices.

2.3 DDI / DKI Changes

As shown in Figure 1, two standard operations in the DDI/DKI [Sun92] have been changed to recognize two additional command parameters and three operations have been added experimentally to the DDI/DKI. The changes allow drivers to interact indirectly with the `pm-driver`, so different power management pseudo drivers can be provided for different hardware platforms, or no pseudo driver if power management is not desired. The changes also permit the implementation of other platform specific power management modules to provide ad-hoc power control for devices which were not initially designed with power control.

The standard DDI/DKI operations `xx_detach()` and `xx_resume()` have two additional command parameters added to their command set, `SUSPEND` and `RESUME`. The `SUSPEND` command parameter requests that the driver save its state, block any further IO, and allow all outstanding IO requests to complete. When it returns, there should be no outstanding requests for IO so that nothing is lost when the device is power cycled. The `RESUME` command parameter restores the driver's state for a power on. As indicated by the grey arrows in the figure, recognition of these commands by the driver is optional.

`ddi_dev_is_needed()` is a new operation called by the device driver when a component of a device is needed at a certain power level. It returns when the component is at the requested level. `xx_power()` is a new operation that is implemented by the device driver. It takes one command parameter, indicating whether the power should be turned on (`ON`) or off (`OFF`), and is called by `pm_driver` to request that the device driver turn the power on or off. `ddi_power()` is a new operation that is called by a device driver which doesn't want to handle power management itself to request that the kernel handle power management with the system default. The black arrows in Figure 1 indicate that the driver is required to implement these.

3. System State Checkpoint and Resume

A major component of the nomadic enhancements is system checkpoint/resume (CPR). A system *checkpoint* saves the state of the entire system, including user-level processes, to nonvolatile storage. A system *resume* restores the system to the checkpointed state at a later time. The primary benefit of CPR for nomadic computing is its role in power conservation. Automatic checkpointing when the battery is low protects the user from undesirable state loss. CPR also matches the people's expectations about how a nomadic machine should be used. With CPR, a machine can be moved from one place to another without going through the time-consuming bootstrap process. Resume times are typically on the order of a minute or less.

CPR has been available in MS-DOS and on a few portable Unix machines running SunOS 4.1 for some time. However, the SVR4 system is considerably more complex than MS-DOS or SunOS 4.1, so the design issues involved are somewhat more challenging. The design goals for CPR in SunOS 5.x are the following:

- CPR should provide extremely reliable service,
- System administration and maintenance should be as simple as possible,

- The design should remain flexible and extensible so it will easily work with new releases of SunOS 5.x,
- The checkpoint and resume times should be as short as possible, to encourage people to use the facility.

There are two possible design strategies for CPR:

- Forcefully shut off all subsystems and take a snapshot of all kernel and user-level process memory images,
- Ask each subsystem to stop and allow the system to save itself.

Most existing implementations of CPR take the first approach [Tadpole93]. The first approach is faster but the outcome is less deterministic. For example, an Ethernet controller might be right in the middle of accepting a packet, and forcing it to shut off causes that packet to drop. For this reason, our design uses the second approach. Figure 2 schematically illustrates the checkpoint process, and the next two subsections examine CPR in more detail.

3.1 System Checkpointing

User processes are the first system component to be checkpointed. When a checkpoint is initiated, all user processes are requested to stop. This is accomplished by sending a new user signal, `SIGFREEZE`, to each process. The default action of `SIGFREEZE` is to stop the process. Applications that need a chance to perform certain housekeeping tasks prior to the checkpoint (e.g. close down network connections, notify servers or clients of impending unavailability) can catch the signal. Once the user processes are frozen, all modified pages with backing store are paged out, and all such pages are invalidated and purged from the page cache. User-level processes are restored from these pages on resume.

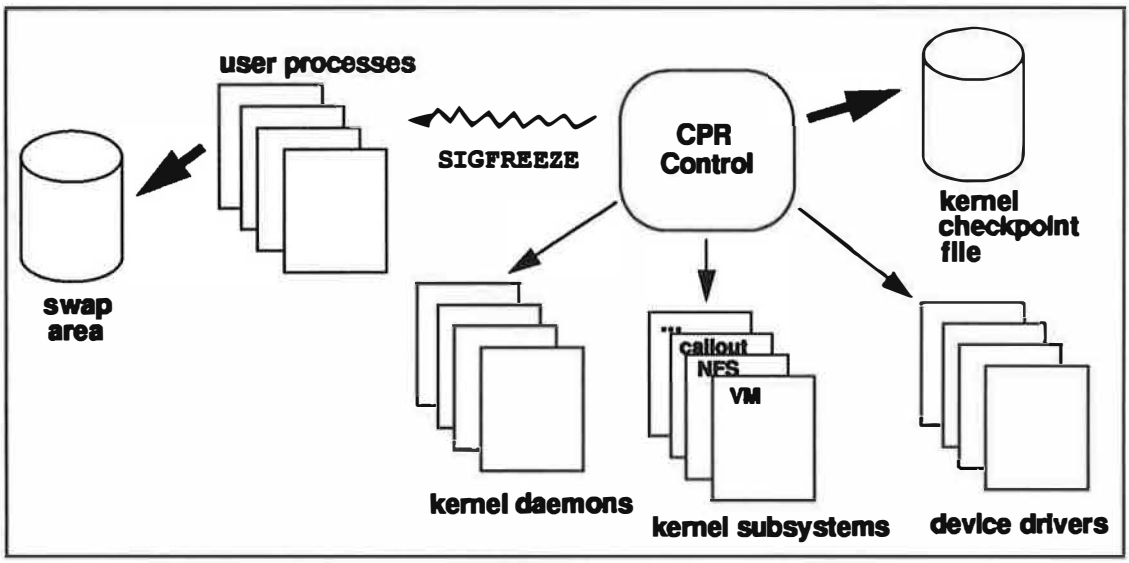


Figure 2 System Checkpoint Process

After all user processes are stopped and their pages are pushed out to the swap device, the kernel daemons are requested to stop. Since each daemon is unique, it needs to stop at a different location. A callback mechanism allows each daemon to specify where it should be stopped. A callback handler can be registered with the CPR callback mechanism by the daemon at thread initialization time if the daemon requires special action prior to checkpointing. The installed handler is part of the daemon and uses the same synchronization mechanisms to stop the daemon.

After all the user processes and kernel daemons are stopped, no further base level activity is possible. The next step is to give all interested subsystems (e. g. VM, callout processing) a chance to pre-

pare themselves for checkpoint. This is accomplished via the same callback mechanism used for stopping the kernel daemons. Only those subsystems that require special processing prior to checkpointing need register callbacks.

Finally, all device drivers are suspended. Although theoretically it would have been possible to design checkpointing so that IO in progress need not complete, the probability that a restarted IO request would succeed on resume is rather low. Therefore, the checkpoint subsystem blocks all new requests and the kernel waits for all outstanding IO requests to complete. After the IO queue is empty, the driver saves the device hardware and software state to data structures in memory using the same suspension mechanisms as power management suspension. Once all the device drivers are stopped, interrupts are turned off and the system enters a completely dormant state. All valid kernel pages are written out to the kernel state file on the root device (e.g. disk or network) and the checkpoint process is complete.

3.2 System Resume

On the resume side, the kernel loading program started by the PROM detects a resumable system by an implementation dependent mechanism and loads in a special resume module. This resume module reads the kernel state file, and restores the kernel memory image and the MMU back to the checkpointed state. Control is then transferred back to the kernel, and the reverse of the checkpoint sequence is executed. When user level processes are ready to run, a `SIGTHAW` signal is sent, in case they need to initialize before beginning normal execution.

4. PCMCIA on Unix

Unlike desktop machines, portable computers have a limited amount of cabinet space for expansion slots and a limited amount of power for peripheral devices. In response to the need for a smaller form factor and a lower power bus, portable computer manufacturers formed the Portable Computer Manufacturers' Card Interface Association (PCMCIA), and developed a new bus standard, having the same name [PCMCIA93]. The PCMCIA standard defines the physical form factor, electrical bus interface (8 or 16 bit), and a software API for small, credit-card-sized devices. The standard allows mass storage, IO and memory peripherals to be used on multiple different hardware system architectures and operating systems. Although the software interface in the standard was developed specifically with implementation on the Intel x86 architecture and the MS-DOS operating system in mind, any computer which implements a bus interface conforming to the PCMCIA mechanical and electrical specifications can utilize a PCMCIA card. Since the existence of the standard is catalyzing volume manufacture, implementing software support for PCMCIA on Unix would provide the Unix user with access to a wide variety of previously unavailable low power peripherals.

The goals of the PCMCIA implementation are the following:

- Provide a software architecture and API that adheres closely to the PCMCIA standard without violating the architectural assumptions of the SVR4 device driver architecture,
- Supply enhancements in areas where the standard is currently underspecified or lacking,
- Assure that the implementation supports maximum portability of PCMCIA card drivers between SunOS on different hardware platforms (e.g. SPARC and Intel x86),
- Feed the results back into the PCMCIA standards process to grow the current software interface beyond its focus on MS-DOS.

In the following subsections, we describe the implementation of the PCMCIA 2.01 standard API on SunOS 5.x.

4.1 The Card and Socket Services Design in Unix

The PCMCIA standard specifies the *Card Services API* as an interface between a device driver for a card and the system software layer managing the PCMCIA bus controller hardware, or *adapter*.

The adaptor provides the physical slot, or *socket*, for the PCMCIA card and a hardware interface between the PCMCIA bus and the system bus. The Card Services API allows a card device driver to control the adapter independently from the specific adapter hardware. The *Socket Services API* is a hardware-independent interface between Card Services and the software layer that controls a particular adapter. It is possible to have multiple Socket Services layers to support multiple different types of adapter hardware.

Figure 3 illustrates the PCMCIA software driver architecture in SunOS 5.x and its mapping onto the hardware. In the figure, grey rectangles are hardware modules and grey arrows indicate interaction between software components and hardware. Black arrows indicate interaction between software components. In the SVR4 architecture, device drivers for bus controllers such as a PCMCIA adaptor are called *nexus drivers* [Sun92]. Drivers for devices on the bus are children of the nexus. Figure 3 indicates parent-child relationships with a striped line. The Socket Services layer corresponds to a nexus driver in the SVR4 architecture, with a separate nexus driver required for each different type of adapter hardware. A particular PCMCIA nexus driver is the parent of all card drivers for cards in that adapter's sockets, and, in turn, the PCMCIA nexus is the child of the system bus nexus. Unlike other device drivers, a PCMCIA card driver does not interact directly with its parent nexus. Rather, as specified by the standard, the card driver calls into the Card Services layer, and the Card Services layer deals with the nexus.

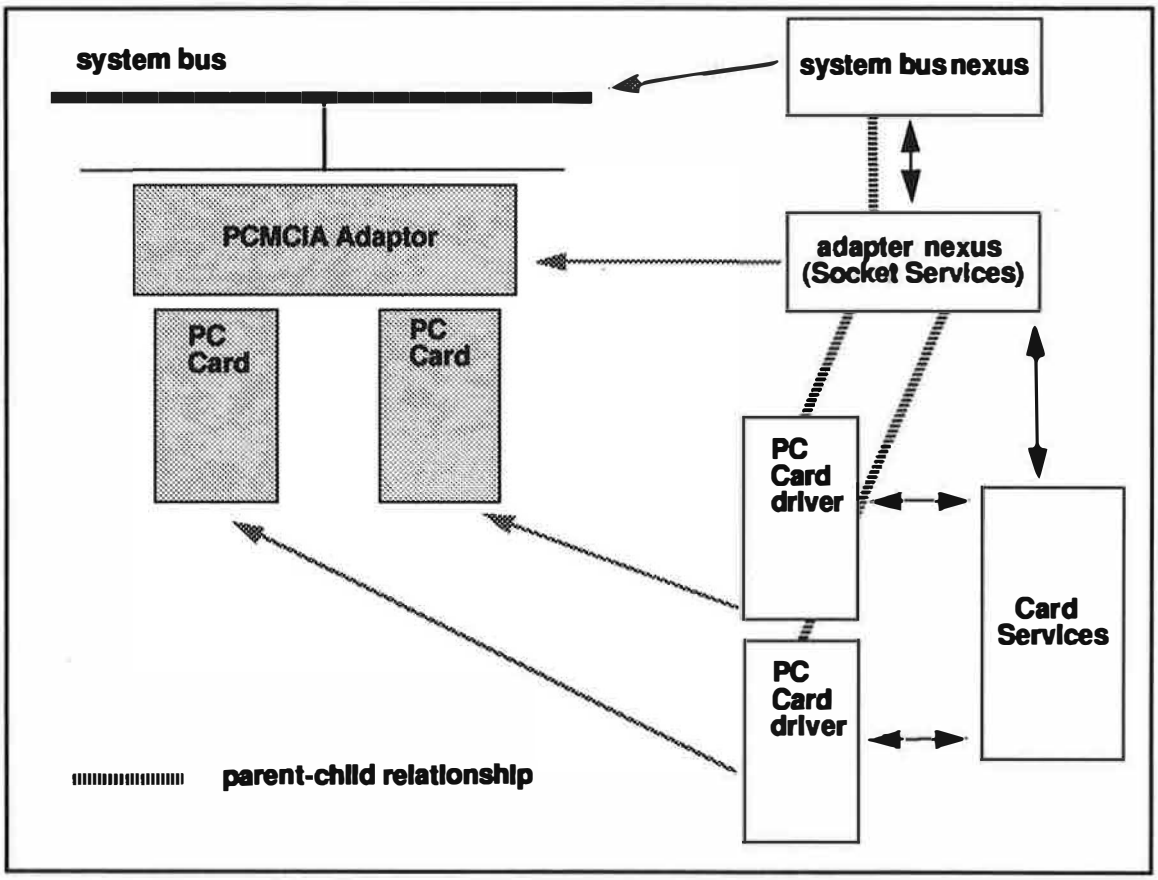


Figure 3 PCMCIA Hardware and Driver Software Architecture

The PCMCIA standard also specifies calls in the Card Services API allowing a device driver to load and unload a Socket Services implementation. The mapping of Socket Services to a parent nexus driver in SunOS 5.x means that this feature cannot be provided on the SunOS 5.x implementation of PCMCIA. Allowing a child to replace its parent nexus is not permitted in the SVR4 device driver architecture.

The SunOS 5.x implementation of Card Services is a collection of function entry points in the kernel. These functions are located in a miscellaneous kernel module that is not part of the device

driver hierarchy. The module handles all Card Services requests from all PCMCIA device drivers and makes the calls into the appropriate parent nexus. The Card Services functions require the driver to pass a client handle that uniquely identifies the requesting driver instance. SunOS 5.x Card Services implements this client handle as a pointer to a structure containing the driver's per-instance information. Card Services uses this handle to locate the appropriate parent nexus in an implementation specific way. In the PCMCIA standard, the interface between the parent and the child is publicly specified in the Socket Services interface. In the SunOS 5.x implementation of PCMCIA, this interface is private and not available for use by the device driver itself. However, the interface is available for developers of PCMCIA bus adapter hardware who need to develop a custom nexus.

4.2 Modifications to the Card Services Interface

The specification of the current PCMCIA Card Services API is highly tuned for the MS-DOS operating system. The standard specifies that Card Services has a single function entry point through which all Card Services requests are vectored. This single entry point is specified to require the same number of arguments no matter which Card Services function is being requested. The arguments specified in the standard are of fixed type, and different type arguments for different Card Services calls must be packed as untyped bytes into a variable length byte array. In the current PCMCIA standard, each processor type, processor mode (protected, real, etc...), subroutine call method (near, far) and operating system on which Card Services is implemented specifies an implementation-specific binding that dictates the details required to make the call. The model is very similar to an MS-DOS INT 21 function call.

This design hinders driver portability for operating systems, such as SunOS 5.x, that run on multiple system architectures and multiple processors. As part of Sun's efforts in the PCMCIA standard process, a C language calling convention is being proposed. The proposal implements a single Card Services function entry point with a variable argument list of pointers to function-specific structures and/or opaque data types, somewhat like an `ioctl()`. Details such as the sizes of various address and data types, byte ordering for shared structure members, and generic system resources managed by the kernel are kept hidden from the driver developer. Selective data hiding allows a device driver to be source compatible between SunOS 5.x on the SPARC architecture and SunOS 5.x on the Intel x86 architecture. Certain differences between the SPARC and Intel x86 architectures, such as card IO port accesses, are hidden using register access macros supplied by the PCMCIA header files. On SPARC, card IO registers are mapped into the device driver's virtual memory address space, while on the Intel x86, the registers are accessed using privileged I/O instructions, but the macros hide this detail from the driver writer.

4.3 The Card Information Structure (CIS) Interface

One of the most important features of the PCMCIA standard is the specification of a mechanism making every PCMCIA card selfidentifying. Every PCMCIA card has an area in which card information is stored, called the *Card Information Structure* (CIS). The CIS is a singly-linked-list of variable-length tuples. Each tuple has a one byte code describing the tuple type, and a one byte link which is the offset to the start of the next tuple in the list. The CIS provides a wide variety of information about the PC card. Each tuple can contain subtuples that elaborate on the information provided by the parent tuple. Although the PCMCIA standard originally only specified enough bits for 256 tuples, as the standard grew, the need for more arose. Since no standard extension mechanism has been proposed, extensions have arisen in an ad-hoc manner.

The only method specified by the PCMCIA standard for dealing with CIS structures is a set of Card Services functions that return the raw contents of a tuple. The device driver is required to take care of tuple parsing itself, even for tuples whose structure is fixed by the standard. The SunOS 5.x Card Services implementation provides the standard Card Services functions, however, it also includes a tuple parser that parses all tuples specified by the standard into structures defined in the PCMCIA header files. A device driver need not include any tuple parsing code itself unless information from a nonstandard tuple is required.

PCMCIA CIS structures are also used by the PROM during booting to build the kernel device tree on SPARC systems. SPARC systems contain a rudimentary CIS interpreter and tuple parser in the body of the boot PROM. This CIS interpreter knows about certain common tuples that provide device identification and configuration information, and the interpreter and parser build simple nodes in the kernel device tree for each card found at boot time. In addition, card developers can include Fcode, the programming language used by Sun's boot PROM for selfidentifying devices, in the CIS of their cards, allowing the boot PROM to configure PCMCIA cards just as Sbus cards are configured currently. For cards that appear to the system as mass storage or network devices, the PROM can load boot images or establish network connections, allowing the PROM to boot the SunOS 5.x kernel from files read off PCMCIA mass storage or network devices. In SunOS 5.x for Intel x86 systems, using a PCMCIA card as a boot device is not yet supported, since the boot PROM on Intel x86 machines is not programmable.

5. Serial Wide-Area Connectivity and Link Management

Traditionally on the desktop, the primary connectivity option available to users has been 10 Mbps Ethernet. The most common form of network connectivity available to nomadic users currently is a simple serial modem connection (2400 to 14.4K bps, faster with compression) via an analog telephone line. More recent innovations in mobile connectivity have made alternate options available, such as spread-spectrum and packet radio, cellular, and satellite. One characteristic common to most nomadic connectivity options regardless of their bandwidth and latency characteristics is point to point serial transfer. Support for efficient serial connectivity is therefore essential for nomadic users.

The goals of the serial connectivity design are the following:

- The serial connectivity protocol should slip easily in at the ISO data link layer, and use standard higher level and lower level interfaces,
- Performance of the protocol over low bandwidth, high latency serial networks, such as analog telephone, should be adequate,
- The serial protocol should be accommodated by existing network infrastructure (e.g. routers),
- Connection management should optimize connection time, so that expensive or unreliable connections need not remain active when they are not in use,
- The protocol should contain proper support for security.

These goals lead to the incorporation of the Point to Point Protocol (PPP) [Simpson92a] [Simpson92b] [McGregor92] [Lloyd92] into SunOS 5.x, and the development of the asynchronous link manager. The next two subsections describe these new system components.

5.1 Point to Point Protocol (PPP)

Most network applications built for the desktop environment have been engineered for the Ethernet, and specifically for the TCP/IP or UDP/IP protocols. For nomadic users to be able to employ standard network applications, like the Network File System (NFS) over a serial line, the same kind of higher level services available on Ethernet, e.g. TCP/IP and UDP/IP, are required over a serial connection. PPP is a standard protocol developed by the Internet Engineering Task Force Network Working Group for point-to-point links such as dial-up modem servers [Simpson92a] [Simpson92b] [McGregor92] [Lloyd92]. With PPP, applications can access the wide-area serial connection through the same system interfaces and higher level protocols used for the local area network.

The PPP protocol consists of three components:

- An encapsulation method for datagrams over serial links,
- A Link Control Protocol (LCP) for the establishment and configuration phases of starting a connection, and for testing,

- A family of Network Control Protocols (NCP's) for connecting over different network layer protocols.

The PPP design specifies an encapsulation protocol over bit-oriented synchronous links and asynchronous links with 8 bits of data and no parity. The links must be full duplex, but can be either dedicated or circuit-switched. An escape mechanism is specified to allow control data such as software flow control (XON/XOFF) to be sent transparently, and to remove control data interjected by intervening software or hardware. Only 8 additional octets are necessary for encapsulation, and the encapsulation can be shortened to 2 octets if bandwidth is at a premium, supporting lower bandwidth connections. The encapsulation scheme provides for multiplexing different network-layer protocols over the link at the same time. As a result, PPP can provide a common solution for easy connection with a wide variety of existing bridges and routers. Finally, PPP provides two protocols for authentication: a password authentication protocol and a challenge-handshake authentication protocol.

5.2 The Link Manager

The asynchronous serial line link manager is a user level daemon that takes care of maintaining serial line connections. It automates the process of connecting to a remote host when PPP service is established. The connection can be initiated either by simply sending a IP datagram to a disconnected host, or by receiving notification from a remote host that a connection is desired. In this way, network services such as NFS can be provided without requiring a continuous, physical connection. In Figure 4, the position of the link manager in an experimental serial line architecture is shown. The link manager is normally started at boot time by `init`. It reads a configuration file, builds internal data structures, opens up a connection to the kernel IP-Dialup module (`ipdcm` in the figure), creates and opens the FIFO on which it listens for incoming (dial-in) requests, and then enters an idle state waiting until a message appears on either of the opened connections.

An outbound connection is initiated when the link manager receives a connection request message from IP-Dialup. It checks to see if the connection request corresponds to a configured "path", consults the UUCP data base files for modem and destination system information, and then places a phone call to the destination host. After the physical link is established (i.e. the two hosts are connected over the phone line), the link manager configures and initiates PPP. Once the data link layer is established and the PPP modules on the peer hosts are communicating with each other, the link manager starts IP and, if specified, modifies the local route table to indicate that the newly created path is to be used as the default route. The link manager then passively monitors the connection until an event such as idle time out, line disconnect, or an error condition occurs. When this happens, it will disconnect from the peer, clean up external data structures like the route table, and return to the idle state.

An inbound connection is initiated when the link manager receives a notification that the login service has opened the named pipe FIFO. The login service is a separate process that is invoked by the `login` program. When the login service is invoked, it opens the named pipe FIFO and passes the login name and the file descriptor for its standard input to the link manager listening on the other end of the named pipe. It then waits until the pipe between it and the link manager is closed, indicating a disconnect, then exits. This allows `login` to regain control over the serial port again and display the login prompts. When the link manager is notified that the login service has connected to the named pipe FIFO, it reads the information written to the pipe, checks to see if the login name corresponds to a configured path, and then configures and initiates PPP. The rest of the scenario is the same as that described for outbound connections.

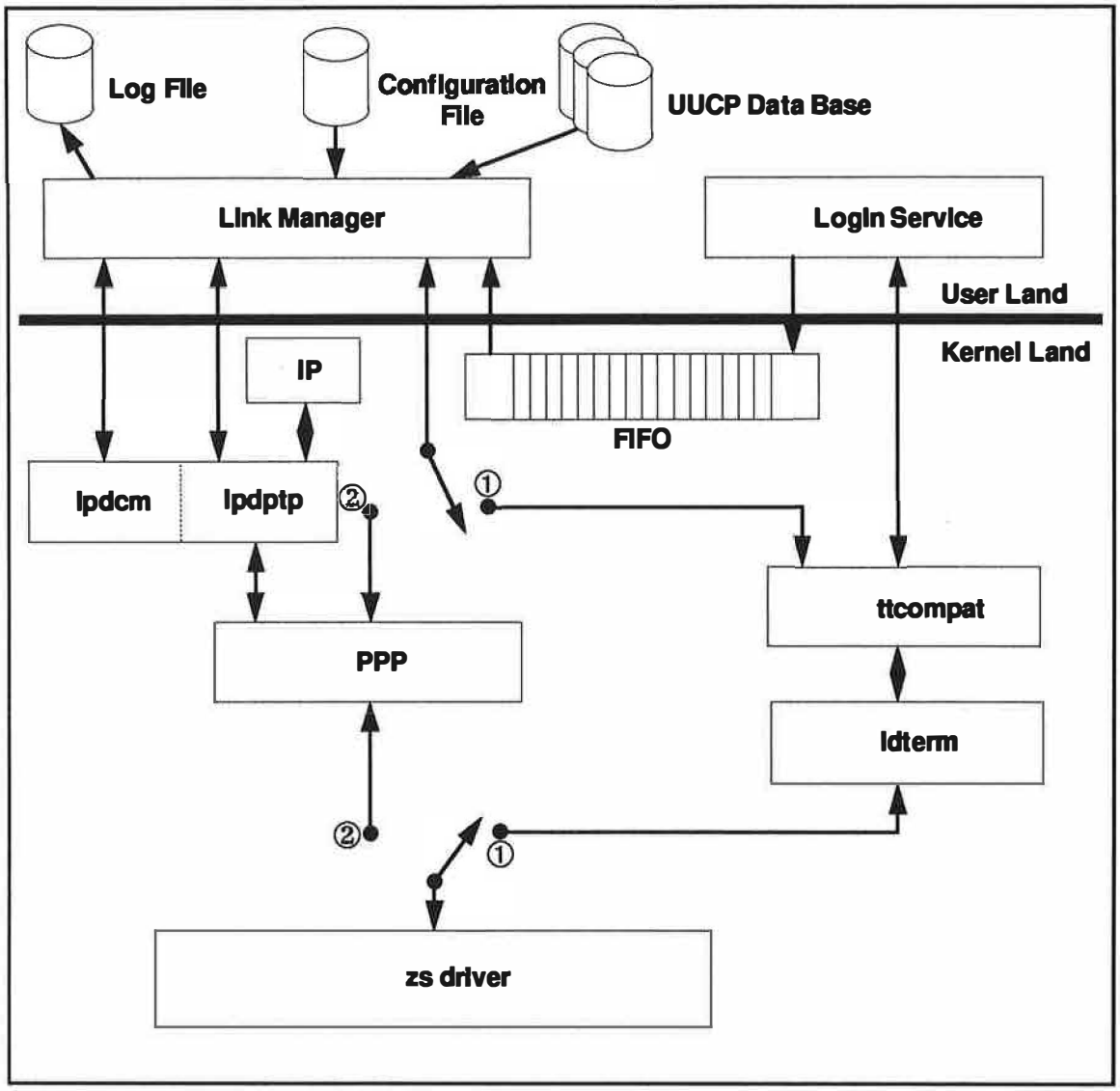


Figure 4 Components and Interfaces in the Link Manager

6. Nomadic Electronic Mail

Electronic mail is the canonical example of an attractive mobile application. The asynchronous nature of electronic mail allows a mobile user to send email from one location and get the reply at another. Mobile users require the same level of electronic mail service as their deskbound counterparts. In addition, since a wide variety of wide-area and local-area connectivity options of varying cost and performance are likely to be available to a mobile user, nomadic access to electronic mail should reduce the need for connectivity unless it is absolutely required, and allow good access even over high latency, low bandwidth connections. For many nomadic users, remote mail may be the single most convenient way of maintaining electronic connectivity, supplanting terminal emulators and file transfer which are the current most widely used options.

Based on these considerations, the following goals for the nomadic electronic mail application were defined:

- Access to electronic mail should be possible over any wide-area or local-area connection,
- It should be possible to manipulate mail state while disconnected from the network, so a user can read mail without requiring a potentially costly wide-area network connection,

- The mail application should be usable even after an unanticipated disconnection from the network or a server failure,
- A user should be able to indicate that they need a certain level of functionality when they are disconnected and the mail application should accommodate this request,
- The nomadic electronic mail application's user interface should differ from the standard desktop UI only in those areas where specific nomadic problems must be addressed.

The following two subsections describe design solutions for these goals.

6.1 IMAP Mail Protocol

The underlying mail protocol used by the electronic mail system plays an important part in determining the connectivity characteristics of the electronic mail application. A variety of currently popular mail protocols and other ways of remotely viewing mail (POP [Rose91], PCMail [Lambert88], remote X [Scheifler92]) were examined and the Interactive Mail Access Protocol (IMAP) [Crispin90] [Crispin92] was selected. The primary reason for selecting IMAP is because it was designed as a interactive protocol rather than a protocol for simply moving messages from one message store to another. The IMAP protocol makes efficient use of the underlying connection bandwidth by breaking the message into logical chunks, allowing the mail reader to request only the parts of the mail message that the user really wishes to see. For example, if a mail message is in a multimedia format such as MIME, an IMAP-based mail reader could only request the text of the mail message and allow the user to make a choice as to whether or not they wanted to wait for a multimedia attachment to be downloaded.

Although IMAP was a good starting point, its main use to date has been over standard Ethernet and thus its implementation and definition required improvement to support mobility. For example, IMAP sends an envelope structure over the connection for each message, even though the envelope structure is unneeded unless the user actually indicates a desire to reply. To increase the efficiency of IMAP over low bandwidth, high latency wide-area connections, the protocol was modified to reduce the amount of such unnecessary data sent across the connection. In addition, IMAP originally did not support disconnection. Disconnected operation is handled by calculating a checksum on the server to determine if some other mail reader has modified the section of the remotely cached file. Although the checksum should be updated every time an operation affecting mail state is executed, transmitting the checksum separately could cause much higher packet traffic. By piggybacking the checksum on the completion reply packet, the update information is sent as part of the standard request/reply protocol, eliminating the need for a separate request/reply to handle the checksum. At the moment, the mobile IMAP does not support full functionality while disconnected, however. For example, full support of searches for text in the body of a message is impossible unless the body of the message is local or the connection is up.

6.2 Nomadic User Interface Enhancements

For the most part, the user interface of a nomadic mail reader application need differ little from one for desktop use. Indeed, close user interface compatibility between the mail reader for nomads and for the deskbound means that users need not learn a new command interface when they take their machine on the road. *ROAM* is a prototype nomadic mail reader similar in basic function to the standard Sun reader *Mailtool* but with a Motif interface. Besides those places where Motif and OpenLook differ, the command set is almost the same as *Mailtool*.

The exceptions are where additional command support for remote or disconnected operation is required. A command interface allows messages to be downloaded for caching on the local machine, so that users can view mail while disconnected. Users may also want to know which messages are local and which are remote, to avoid an unpleasant surprise while disconnected when they attempt to read a message which is not there. Headers of messages that are not locally cached appear differently from those that are only available over the network, and message bodies that are not local are displayed as half full icons. Locally cached messages have their headers displayed in roman font, while remote messages are in italic. Another user command allows the mail

queue to be manipulated, so that unsent messages can be deleted if desired. Finally, users who are connected over wide area serial links need the ability to specify more stringent searches to reduce the amount of traffic and number of misses when trying to find a particular message. Detailed search capability was added including search by dates, text in message body, status of messages and many others. Using this search capability, a user could search for a message received this week concerning a certain topic eliminating the large number of messages that would appear if all messages received for this topic were searched. The search is then run on the server, rather than downloading all message bodies to the client and running the search locally.

7. Summary

Fostering a more nomadic usage model in Unix requires changes throughout the entire system, from the kernel to the commands of user applications. The primary and most important change in the kernel is automation of power monitoring and control, to reduce the drain on batteries. Power management increases battery life, allowing users to work longer without having to plug into wall power or change batteries. System checkpoint/resume provides ease of use in such areas as quick start and reliability. With system checkpoint/resume, a mobile user can quickly shut down operation at one physical location and start up at another without requiring a lengthy initialization.

The new PCMCIA standard presents an opportunity for Unix-based machines to leverage off of a wide variety of new communication, IO, and storage devices. Implementing the standard in Unix has required a variety of changes, since PCMCIA was originally developed for MS-DOS. The changes map concepts such as Socket Services onto components of the existing SVR4 device system design. Mostly, however, the PCMCIA Card Services layer has been maintained intact, providing high fidelity to the standard in the part of the API which matters most to device driver writers.

Serial line connectivity is and will remain the primary means for nomadic users to tap into the Internet. The PPP protocol, developed by the Internet Engineering Task Force, is an efficient serial line protocol allowing connection with existing network infrastructure. Since serial connections are generally more expensive and less reliable than Ethernet, connection management is required. The asynchronous serial line link manager hides the details of making serial connections, so that a serial connection looks exactly like a standard TCP/IP connection to an application.

While many existing network applications can be used unchanged in the nomadic environment, others may require modification. Electronic mail is an example of the canonical nomadic application, since email is so useful to nomadic users. The IMAP interactive mail protocol, modified for nomadic use, provides the basis for efficient support of email over a serial line and for disconnected operation. The changes made to IMAP are a useful prototype for how application-specific protocols can be changed to support a nomadic model. In addition, modifications to the user interface allow users to specify caching upon disconnection, and provide a variety of other command features specific to the nomadic environment.

8. References

[Crispin90] Crispin, M. "Interactive Mail Access Protocol - Version 2," Internet Engineering Task Force RFC 1176, 1990.

[Crispin92] Crispin, M., "IMAP2BIS - Extensions to the IMAP2 Protocol," Internet Engineering Task Force RFC 1176, 1992.

[Lambert88] Lambert, M., "PCMail," Internet Engineering Task Force RFC 1056, 1988.

[Lloyd92] Lloyd, B., and Simpson, W., "PPP Authentication Protocols," Internet Engineering Task Force RFC 1334, 1992.

[McGregor92] McGregor, G., "The PPP Internet Control Protocol (IPCP)," Internet Engineering Task Force RFC 1332, 1992.

[PCMCIA93] *PCMCIA Standard Release 2.01*, Portable Computer Manufacturers' Card Interface Association, 1993.

[Rose91] Rose, M., "Post Office Protocol - Version 3," Internet Engineering Task Force RFC 1081, 1991.

[Scheifler92] Scheifler, R.W., and Gettys, J., *X Window System*, Digital Press, 1992.

[Simpson92a] Simpson, W., "The Point-to-Point Protocol (PPP) for the Transmission of Multi-protocol Datagrams over Point-to-Point Links," Internet Engineering Task Force RFC 1331, 1992.

[Simpson92b] Simpson, W., "PPP Link Quality Monitoring," Internet Engineering Task Force RFC 1333, 1992.

[Sun92] *SunOS 5.1 Writing Device Drivers*, SunSoft, Part No. 801-2871-10, 1992.

[Tadpole93] "The Nomadic Computing Environment", Tadpole Corp., 1993.

A Mobile Networking System based on Internet Protocol(IP)

Pravin Bhagwat
pravin@cs.umd.edu
Computer Science Department
University of Maryland
College Park, MD 20742

Charles E. Perkins
perk@watson.ibm.com
IBM, T.J. Watson Research Center
Hawthorne, NY 10562

Abstract

Due to advances in wireless communication technology there is a growing demand for providing continuous network access to the users of portable computers, regardless of their location. Existing network protocols cannot meet this requirement since they were designed with the assumption of a static network topology where hosts do not change their location over time. Based on IP's Loose Source Route option, we have developed a scheme for providing transparent network access to mobile hosts. Our scheme is easy to implement, requires no changes to the existing set of hosts and routers, and achieves optimal routing in most cases. An outline of the proposed scheme is presented and a reference implementation is described.

1 Introduction

In the last two years, we have witnessed two major changes in computer technology. First, portable computers which are as powerful as some desktop workstations in terms of computing power, memory, display, and disk storage, are beginning to appear. Second, with the availability of wireless adapter cards, users of laptop computers are no longer required to remain confined within the wired LAN premises to get network access. Users of portable computers would like to carry their laptops with them whenever they move from one place to another and yet maintain transparent network access through the wireless link. By transparent network access we mean the ability of a mobile user to setup and maintain network connections despite migration from one network to another. This movement may possibly introduce a momentary pause in the operation, but it should not require reinitialization of network connections. The existing set of network protocols do not meet this requirement since they were designed with the assumption of a stationary network topology where hosts do not change their location over time.

The problem of providing continuous network connectivity to mobile computers has received considerable attention [3, 12, 13, 14], especially in the context of networks based on the TCP/IP [10, 11] suite of protocols. The proposed solutions either require changes to the existing network architecture [13] or introduce new encapsulation protocols [3, 14] to handle this problem. In contrast, our approach [5, 6], which is based on the use of an existing IP option, does not introduce any new protocol and achieves optimal routing. The solution is completely transparent to transport and higher layers and does not require any changes to stationary hosts and routers.

We have implemented our scheme on a set of IBM PS/2 model 80s running AIX version 1.2. In this paper, we present an overview of our scheme and provide some details of our current implementation. The details provided in this paper along with references [5, 6] can serve as a guide for interested readers who would like to add mobile networking features to their network test beds.

2 The mobility problem

The Internet is a large collection of networks which share the same address space and inter-operate using a common set of protocols, such as TCP/IP [10, 11]. It is desirable that the integration of mobile computers within the existing Internet be completely transparent to the transport and higher layers so that users of mobile computers can continue to run existing applications. Any acceptable solution for mobility should inter-operate with the existing infrastructure and not require any modifications to existing host or router software. This goal, however, is quite difficult to achieve in practice. An Internet address can be thought of as consisting of two parts, the network identifier and the host identifier. All hosts residing on a (sub)net are required to have the same (sub)net address. Within a (sub)net all attached hosts have a unique host id. The routing infrastructure uses the network part of the address to route the packet to the correct network. Historically, an Internet address served the purpose of a unique host identifier, but the location information was also embedded in it. When a host moves to a new network, it should acquire a new address. Since the transport layer and the network applications assume that network addresses do not change during the lifetime of a connection, the dynamic assignment of new addresses cannot be done without affecting them. To provide *application transparency* it is desirable to devise a method by which hosts retain their home addresses and continue to receive packets despite their migration from one network to another.

Over the last two years several proposals have been made to address this problem [3, 13, 14]. The scheme proposed by Ioannidis[3, 4] relies on a group of cooperating Mobile Support Routers, which advertise reachability to the same (sub)net. Each mobile host, regardless of its location within a campus, is always reachable via one of the Mobile Support Routers (MSR). When a host sends a packet to a mobile host, it first gets delivered to the MSR closest to the source host. This MSR encapsulates the packet and delivers it to the target MSR which strips the encapsulation header and relays the original packet to the mobile host. This approach is optimized to work within a campus environment and cannot be easily extended to support wide area mobility.

In Sony's proposal [13], a mobile host is assigned a new temporary address when it is attached to a new network. The mapping between the home address and the temporary address of a mobile host is kept in an Address Mapping Table (AMT), which is maintained at the routers. Packets transmitted to the home address of the mobile host get intercepted by some router which holds an AMT entry for the mobile host. An address conversion is performed by the router before the packets are forwarded to the physical location of the mobile host. This method requires modifications to routers and host software and has problems inter-operating with the existing hosts unless so-called 'conversion gateways' are used.

Another proposal to support mobile hosts is from Matsushita[14]. This method is also based on the encapsulation approach. A mobile host is assigned a temporary address when it visits a new network. The packets destined to the home address of the mobile host are intercepted by a Packet Forwarding Server(PFS). The PFS encapsulates the packet and forwards it using the temporary address of the target mobile host. The problem with this method is that routing is always sub-optimal unless the software on all stationary hosts is modified.

Our approach [5, 6] is based on the use of an existing IP option and therefore, does not require any changes to the existing hosts and routers. The key idea is that each packet originating from a mobile host contains enough routing information that can be used by the remote host to send reply back to the source along an optimal path. In the rest of the paper, we first present an overview of our scheme and then describe our implementation.

3 System

Our system involves participation of three types of entities, viz., *Mobile Host (MH)*, *Mobile Access Station (MAS)* and *Mobile Router (MR)*. The networking architecture that we assume is that of a set of MASs connected through a wired backbone. An MAS supports at least one wireless interface and functions as a gateway between the wired and wireless side of the network. Due to the limited range of wireless transceivers, a mobile host can setup a direct link layer connection with an MAS only within a limited geographical region around it. This region is referred to as an MAS's *cell*. The geographical area covered by a cell is a function of the medium used for wireless communication. The range of infrared cells is typically limited to about 20 feet, while that of radio frequency cells could be significantly larger.

Within one campus or administrative domain there could be multiple (sub)networks reserved for mobile hosts. Each (sub)network has a separate router which is referred to as Mobile Router (MR). Unlike other routers, an MR is not required to have an interface corresponding to the wireless (sub)net it serves. If an MR has a wireless interface then it can also function as an MAS. The association between an MH and its current MAS is kept in a Location Directory (LD), which is maintained at the MR.

A mobile host retains its address regardless of which MAS cell it is in. It can start sessions with other hosts (both mobile and stationary) and move into other MAS cells without disrupting any active sessions. The movement of a mobile host is completely transparent to the running applications, except possibly for a momentary pause which may occur while the cell switch takes place. An MH can reside in the cell of only one MAS at any given time. Even if cells of two MASs spatially overlap, an MH routes its outgoing packets through only one of them. An MAS can have multiple MHs in its cell.

We use the term *Correspondent Host (CH)* to refer to the host communicating with an MH. In the following discussion, a stationary correspondent host is also referred to as *Stationary Host (SH)*.

4 Overview of Scheme

Our scheme is based on the use of IP's loose source route (LSR) option. The LSR option provides a means for the source host to supply partial routing information to be used by routers in forwarding the datagram to the destination. A source can specify a list of routers which are to be visited in the specified sequence before the datagram is delivered to the final destination. According to the host requirements document [1], return traffic to the originator of the loose source routed packet is also sent with the LSR option by reversing the route taken by the incoming packets. We use this technique to achieve optimal routing between an MH and a CH.

There are four possible communication scenarios depending on whether the CH is stationary or mobile and, if the CH is mobile, whether the MH and the CH are in the same cell or not. We consider each case separately and show how optimal routes are constructed in each scenario.

4.1 Mobile Host to Stationary Host

An MH, while communicating with an SH, issues packets with the LSR option which specifies that packets should be routed via the MAS serving the MH (see arcs 1 and 2 in Figure 1). The SH sends reply packets with the LSR option containing the reversed route. These packets are first delivered to the MAS which forwards them to the MH. Notice that if the LSR option is not used in the reply packets, then these packets would get delivered to the

router (MR) for the MH's (sub)network (subsequently called the wireless subnet). The MR would eventually forward these packets to the MH; however, the complete path followed by the reply packets would not be optimal in this case.

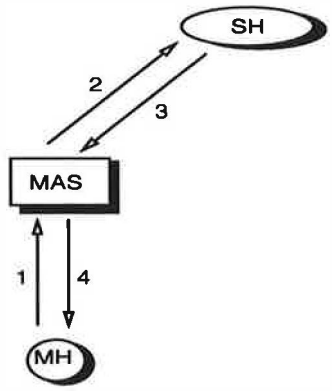


Figure 1: MH to SH

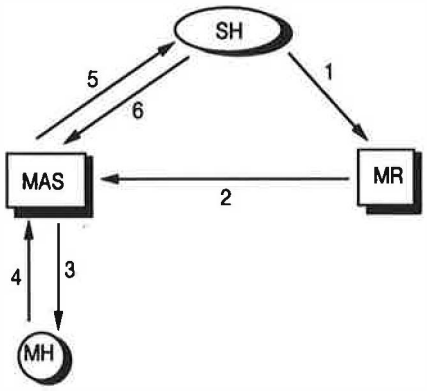


Figure 2: SH to MH

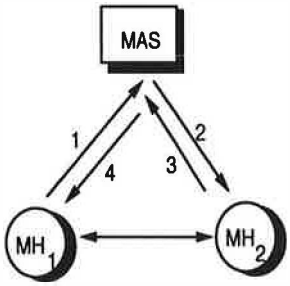


Figure 3: MH to MH (same cell)

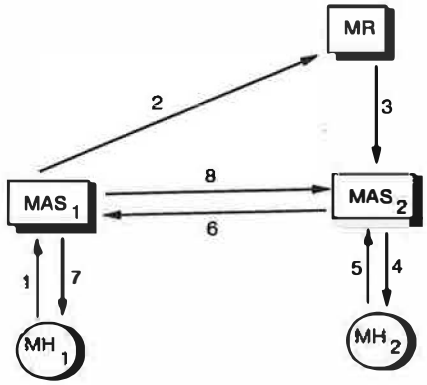


Figure 4: MH to MH (different cell)

4.2 Stationary Host to Mobile Host

An SH need not be aware of the current location of the MH when it initiates a session. If it is not aware, the packet sent from the SH (denoted by arc 1 in Figure 2) arrives at the MR, which advertizes reachability to the wireless subnet. The MR, using the information in its Location Directory, inserts the LSR option in this packet which causes this packet to be delivered to the MH via the current MAS serving the MH (arcs 2 and 3 in Figure 2).

When a reply to this packet is sent, the MH reverses the LSR and sends the packet back to the SH via the MAS (as denoted by arcs 4 and 5 in Figure 2). Once the SH receives a source routed packet, it can send subsequent packets to the MH along the optimal path by reversing the incoming loose source route.

4.3 Mobile Host to Mobile Host within the same cell

An MH does not keep track of other MHs residing in the current MAS's cell. It always uses the current MAS as its default gateway for all outgoing traffic. When an MH initiates a session with another MH, it sends all packets to the MAS just as it would do if it were to send those packets to an SH. Since MAS keeps a list of all MHs residing in its cell, it can forward those packets to the destination MH. If the wireless link layer technology supports

direct MH to MH communication then the MAS can also send an ICMP[9] redirect message to the source MH so that it can directly communicate to the destination MH rather than source routing its traffic through the MAS.

4.4 Mobile Host to Mobile Host in different cells

An MH does not inspect the destination IP address to determine whether the destination host is a stationary host or a mobile host. Consequently, it always starts off by sending packets with the LSR option. By normal routing mechanisms, these packets are forwarded to the MR associated with the destination MH (see arcs 1 and 2 in Figure 4). The MR extends the existing LSR option by inserting the address of the the MAS presently serving the destination MH, and then forwards the packet. Normal routing procedure ensures that these packets get delivered to the MAS serving the destination MH, followed by the destination MH (as shown by arcs 3 and 4 in Figure 4). Notice that the LSR option list of the incoming packet contains addresses of two MASs – one serving the source MH, and one serving the destination MH. The reply packets are sent by reversing the incoming loose source route, which follow the optimal path as denote by arcs 5, 6 and 7 in Figure 4. Once the source MH receives a packet back from the destination MH, it can also send the subsequent packets along the optimal path.

5 Implementation

We have implemented the aforementioned scheme on a set of IBM PS/2 model 80s running AIX version 1.2. Each of these machines is equipped with an infrared (IR) wireless adapter card supporting up to 1Mb/s data transfer rate. The range of an IR transceiver is limited to about 20 feet. This adapter card uses an Ethernet chip. From the perspective of the device driver, a wireless interface behaves much like an Ethernet interface, since both use a carrier sense multiple access (CSMA) protocol. The only difference is that the wireless adapter card does not support collision detection (CD). This shortcoming, however, did not affect us much because most of our experiments were limited to a small cell population.

Our existing implementation consists of approximately 800 lines of kernel code and 1500 lines of user code. It can be thought of as consisting of two parts, viz., the *packet routing part* and the *location information management part.* Actions related to packet routing are performed in the kernel. To avoid creating new data structures, location information is stored implicitly in the kernel routing table. This approach has some obvious advantages. First, minimal kernel modifications are needed to route packets to/from MHs. Secondly, with a little modification, the existing *route* command can be used to manipulate the location information. In the following, we first describe how packets are routed among various components and how location information is managed. Later, a description of the processing required at each component is included.

5.1 Packet Routing

For each MH, which has an address on the wireless (sub)net served by an MR, a host route is maintained by the MR. The current location information of the MH, i.e., the address of the MAS serving the MH is kept in the gateway field of the routing table entry. This routing table entry is distinguished from other entries by the presence of a new flag called `RTF_MOBILE`. Since the MR advertizes reachability to the range of addresses on the mobile (sub)net, an IP packet destined to an MH is first routed to the MR for further delivery. At the MR, one of the host routes with `RTF_MOBILE` flag is chosen to route this packet. The MR knows how to interpret `RTF_MOBILE` flag; that flag specifies that the MR should insert the

LSR option in this packet (by using the MAS[1] as the intermediate hop) before forwarding it. Due to the inserted LSR option, this packet is delivered to the MAS currently serving the destination MH. The LSR option is processed here and, finally, the packet is delivered to the MH.

An MAS performs the forwarding function between the wired and the wireless interface. It keeps a host route corresponding to each MH residing in its cell. Thus, an IP packet which is destined to one of the MHs in its cell is delivered to the MH by the MAS.

Processing of packets originating from MHs is relatively simpler. An MH always keeps a default route to the MAS currently serving it. This routing table entry also has the `RTF_MOBILE` flag set, meaning that the LSR option should be used on all outgoing packets. Packets originating from an MH are source routed via the current MAS and are delivered to the final destination by the normal routing mechanism. The reply packets use the reverse route and are delivered back to the MH by the optimal route.

5.2 Location Information Propagation

We now describe how the location information, which is implicitly maintained by routing table entries at the MR, MAS and MH, is updated when an MH switches cells.

A server program (*beacon_snd*) running on the MAS periodically broadcasts beacons on the wireless interface. These beacon packets contain the IP address of the MAS. An MH, upon entering the MAS's cell, receives these beacons and sets up the MAS as its default router. At the same time the MH also sends an *mh2mr* message to its MR. This message contains the new location information of the MH and a timestamp. A server (*mr_serv*) running at the MR receives this message, uses the location information contained in it to update its routing table, and notifies the previous MAS serving the MH about its migration by sending it a *delete_host_route* message. The previous MAS, upon receiving this message, deletes the host route corresponding to the migrated MH.

All messages are exchanged using UDP packets. The message exchange protocol is very simple and completely implemented by user level processes running at MH, MR and MAS.

5.3 Processing at Mobile Host

The software running on a mobile host performs primarily two functions: cell discovery and insertion of LSR option in all outgoing IP packets.

The purpose of the cell discovery operation is to send a notification and supply the address of the MAS to the network layer whenever an MH performs a cell switch. It is desirable that the cell discovery feature be supported by the link layer. However, due to lack of such support, a user level process (*beac_rcv*) performs this operation at an MH. It continuously monitors beacon packets that are broadcast by MASs, and changes the default route at the MH whenever a cell switch occurs. If the cells of two or more MASs spatially overlap, the MH may potentially receive beacons from multiple MASs. The cell discovery mechanism should avoid rapid cell switching by the MH in this scenario. In our implementation, if beacons from the current MAS continue to arrive at regular intervals, then beacons from other MASs are ignored. The MH switches to another MAS cell, i.e., changes its default route, if it does not receive any beacon from the current MAS for 3 seconds. After an MH has determined the cell to which it belongs, it is required to notify its MR about its current location. This is accomplished by sending an *mh2mr* message to the MR.

[1]The IP address of the MAS currently serving the MH is available from the gateway field of the routing table entry.

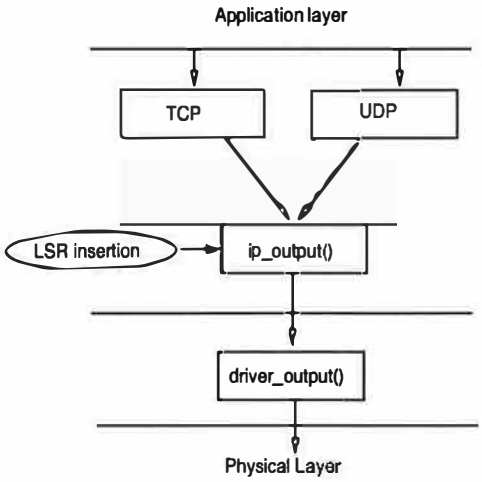


Figure 5: Kernel processing at Mobile Host

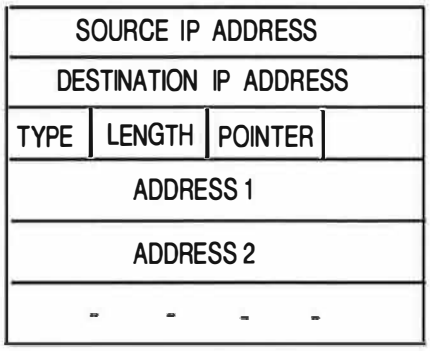


Figure 6: An IP header with LSR option

Unlike cell discovery which is performed by a user level process, LSR option insertion is performed in the MH kernel (see Figure 5). The transport layer (TCP, UDP) running at an MH passes packets to the IP layer for further delivery. The IP layer at the MH is required to modify this packet before passing it on to the network interface. If the outgoing packet does not contain LSR, then conceptually the MH must insert the LSR option and specify the MAS as the only intermediate router. In some cases, the packet passed by the transport layer already contains the LSR option. This option is formed by reversing the route contained in the LSR option of an incoming packet. An existing LSR option in an outgoing packet has already been extended by the MR to include the local MAS as a required intermediate router for the source route. This model of operation conforms to the description given in Comer[2]. However, since in most current implementations all routing is done on the basis of the destination address, the actual operation is somewhat different (see Figures 7 and 8). The only routine which needs modification is `ip_output()`. The changes are described below:

```
if ( pkt does not have LSR option) {
    insert LSR option in pkt;
    first address in option list = pkt.destination;
    pkt.destination = MAS address;
} else      /* option already exists by TCP route reversal */
    if ( pkt.destination != MAS address )
        pkt.destination = MAS address;
```

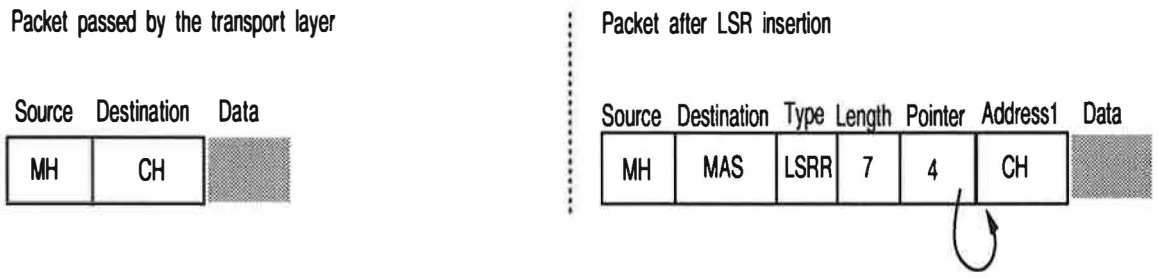


Figure 7: LSR insertion by an MH

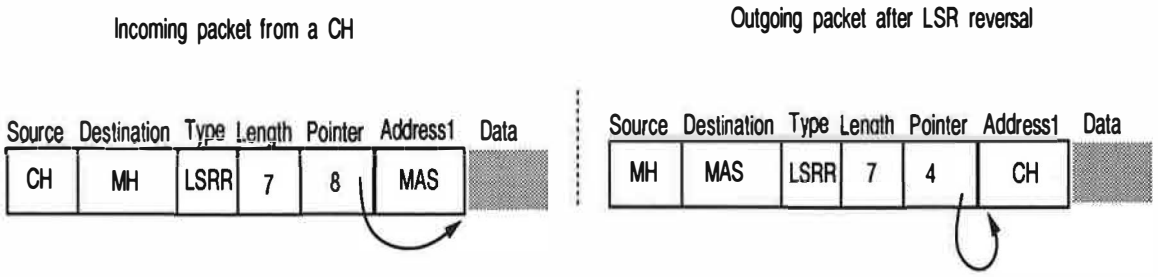


Figure 8: LSR reversal by an MH

Suppose a packet from a CH arrives at the MH and thus has an LSR option. Suppose also that before a reply is sent, the MH switches to another MAS cell. The destination address contained in the reply packet will be that of the previous MAS. Since the MH has already left the previous MAS cell, it is necessary to modify the destination address of this packet to the address of the current MAS. After this packet reaches the CH, the subsequent packets originating from the CH will follow the reverse route and arrive at the MH via the current MAS. Thus, no interruption in the active transport layer session will be observed.

5.4 Processing at Mobile Access Station

The primary function of an MAS is to provide an access point through which an MH, regardless of its address, can attach itself to the network. To forward packets to MHs, an MAS should keep a host route corresponding to each MH that resides in its cell. The host route is installed by the MAS kernel as soon as the first packet (e.g, ARP packet) is received from the MH. Beyond this point, the existing `ip_forward()` code is able to forward packets to the MH via the wireless interface. When an MH moves out of the MAS cell, the corresponding host route at the MAS should be deleted, otherwise the MAS would continue to transmit packets to the MH which no longer exists in its cell. As mentioned earlier, a server process (*mas_serv*) running on the MAS performs the deletion of the host route when it receives a *delete_host_route* message from the MR.

After an MH moves out of the MAS cell, the packets addressed to it may still arrive at the MAS. These packets could be directly forwarded to the MAS currently serving the migrated MH. This, however, would require the MAS to maintain forwarding pointers for all MHs even after they migrate out of its cell, thereby, unnecessarily increasing the protocol complexity and memory overhead. We wanted to design a system where there was no need for any information exchange among MASs. Therefore, packets which cannot be delivered by the MAS are returned to the MR associated with the destination MH. Note that the MAS need not know the address of the MR for this purpose. The MAS only forwards this packet on to the wired side of the network. The normal routing mechanism automatically delivers

this packet to the MR, since it is the router to the MH's home subnet. It is necessary to mark these returned packets from MAS so that MR can distinguish them from other packets. Since there is no extra space available in the packet where this marking can be incorporated, we do it implicitly by setting the last address in LSR equal to the packet destination address.

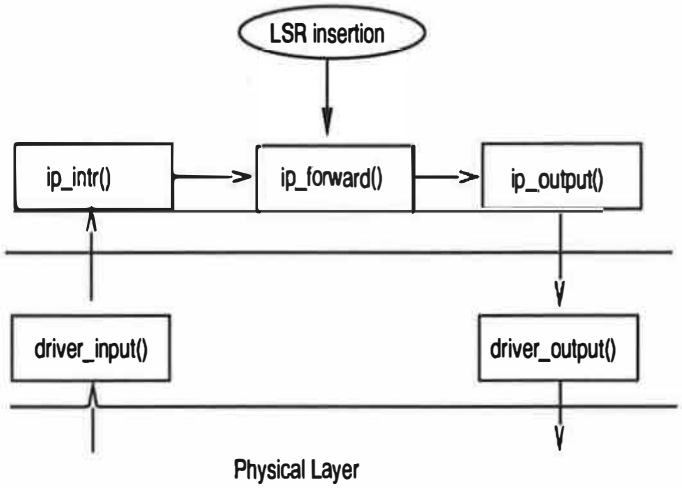


Figure 9: Kernel processing at Mobile Router

5.5 Processing at Mobile Router

An MR performs the following two functions:

1. Maintains location information of all MHs on its (sub)net.
2. Forwards packets to MHs (via an MAS) by inserting/updating LSR option.

Location information is nothing but the necessary routing information which is maintained at an MR. Therefore, the most convenient place for storing it is the kernel routing table. Corresponding to each MH, a host route is maintained at the MR. The gateway field of the route table entry stores the address of the MAS which servers the MH. Each time a MH switches to another cell, it sends a location update message (*mh2mr*) to the MR. A user level process (*mr_serv*) receives this message and updates the routing table by executing a *route* command.

Actions related to LSR option insertion are performed in the kernel. The only routine that needs modification is `ip_forward()`. Packets arriving at the MR, which are destined to some MH, can be classified into four categories depending on the contents of their LSR option. For each case, Figure 10 shows the format of incoming and outgoing packets. The steps taken by the MR are outlined below.

1. Packets originating from Stationary Host : The packet originating from a stationary host, and destined to some MH, usually not contain any LSR option until it reaches the MR associated with the destination MH. The MR modifies this packet (as shown in case 1 of Figure 10) by inserting an LSR option in it. Notice that the original packet destination address is moved in LSR option list and the new packet destination is set equal to the address of the MAS presently serving the MH. This modification causes this packet to be forwarded to the MAS, where the LSR option is processed, and finally the packet is delivered to the MH.

2. Packets originating from Mobile Host : An MR can also receive packets which already contain LSR option. This can happen, for example, when an MH (say MH1) starts up a session with another MH (say MH2). Suppose, in addition, that MH1 and MH2 reside in the cells of MAS1 and MAS2, respectively. When the packet originated from MH1 arrives at the MR, it appears as shown in case 2 of Figure 10. The destination address in the packet header contains the IP address of MH2. The LSR option list contains the address of MAS1 with the pointer pointing beyond the end of the address list.

The MR takes the destination address and appends it to the end of the option list and places the IP address of MAS2 into the destination field of the packet header. Normal forwarding mechanism first delivers the packet to MAS2 and the packet is finally delivered to MH2.

3. Packets returned from Mobile Access Station : A packet arriving at the MR could also be from an MAS which failed to deliver this packet to the target MH. As in the previous case, this packet also contains the LSR option with one address and pointer pointing beyond the end of the address list (see case 3 in Figure 10). However, unlike the previous case, the MR should forward this packet to the current MAS serving the MH without extending the LSR option list. To do that, the MR should be able to distinguish this packet from the packets mentioned in case 2. The MR compares the destination address of this packet against the address contained in the LSR option list. If it is a packet returned from an MAS, then these two addresses are same. In this case, the MR sets the destination address field in the packet to the address of the current MAS serving the MH, resets the pointer in the LSR option to the beginning of the address list, and forwards the packet.

4. Incorrectly processed packets from Stationary Host : When an MH starts a conversation with an SH, the packets arriving at the SH contain LSR option. If the SH correctly processes LSR option, the reply packets will automatically follow the reverse path without any involvement of the MR. Unfortunately, several IP implementations, such as in 4.3 BSD and SUN OS 4.1, do not correctly process LSR option. Even though the incoming source route is reversed and included in the reply packet, the destination address is set equal to the original source and pointer is made to point to the end of the option list. This has the effect of sending the packet straight back to the source, rather than sending it along the reverse path. Since the destination of this packet is an MH, normal routing mechanism delivers this packet to the MR associated with the MH. A sample of such a packet is shown in case 4 of Figure 10.

The MR checks the last address in the LSR option list. If this address is of the MAS associated with the destination MH, then the packet is coming from an SH which does not correctly process LSR options. The MR swaps the addresses stored in the destination address field and LSR option field, resets the pointer to the beginning of the address list, and forwards the packet to the network interface output routine.

5.6 Overhead

Processing and memory requirements of our scheme are very moderate. Overhead of storing LSR option is 8 bytes per packet (and 12 bytes in case of MH to MH communication, when they reside in different cells). Each host route requires 128 bytes of memory as it is stored in an *mbuf*[2]. At an MAS, the number of host routes is equal to the cell population. If we assume 20 as an upper bound on the cell population, only 2K bytes would be required to keep the routing information. At an MR, there will be as many host routes as the number

[2] mbuf is a data structure that is used for dynamic memory allocation in Unix

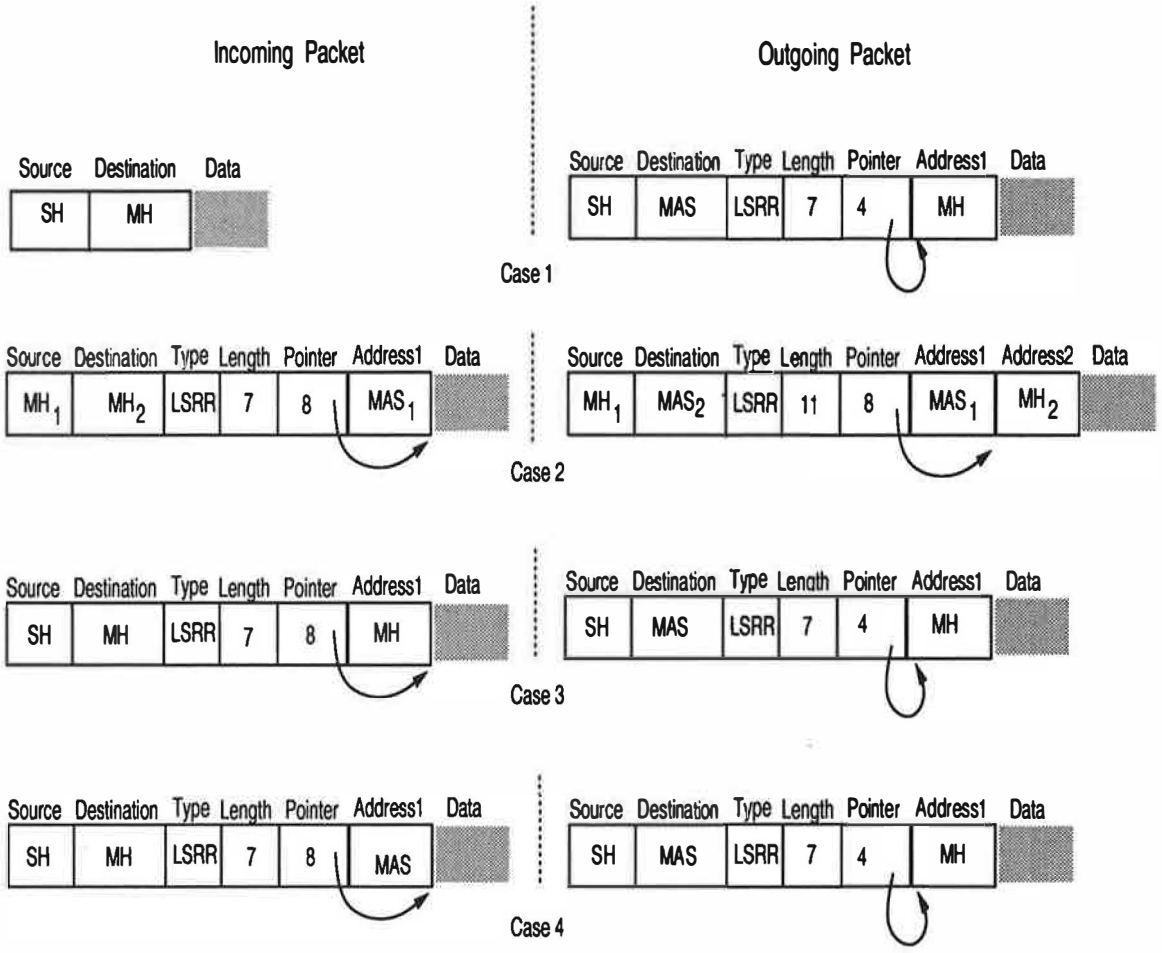


Figure 10: Packet processing at MR

of mobiles which are assigned addresses on the wireless subnet. Suppose an MR serves 256 mobile hosts, then it will require about 32K bytes to store the location information. Given that we only require one MR per wireless subnet, this is a very reasonable figure. It is worth mentioning that it is not necessary to keep the location information in the kernel routing table. We did it only to simplify our implementation effort. A further reduction in data memory requirement can be achieved if a separate table is used to keep the location information, although more processing would be required.

6 Future Work

We currently have in operation the ability to let the MAS do the LSR insertion instead of the MH. With this capability, one could cause the MAS to understand more about which correspondent hosts do correct LSR processing, and limit the LSR insertion to those hosts which correctly process it.

Our modular system enables us to replace the forwarding features based on LSR with similar features using encapsulation, and that capability is being taken advantage in newer systems [7] based on the work described in this paper.

Currently, our location updates are propagated by a special "protocol" called *mh2mr*. This protocol could be easily replaced by a facility whereby the MH pings the MR with the record-route option. Then when the MR receives the ping, it would notice that the sender

was a MH within its mobile subnet, and then the MR could infer that the first stop along the way is the MAS currently serving the MH. Besides eliminating unnecessary protocol, this method has the advantage that the MR has a measure of the time delay associated with the current location of the MH. This information could trigger the instantiation of another MR closer to the MH, which would reduce the delay associated with sub-optimal routes arising from imperfect LSR implementations.

Some measures will have to be taken to avoid the sub-optimal routing currently to be expected from all UDP applications, especially because of the prevalence of NFS. We intend to perform this in conjunction with the previously mentioned capability for providing LSR insertion at the MAS.

7 Concluding Remarks

Alternative solutions for supporting mobility are based on the encapsulation approach. This approach relies on the presence of a packet forwarding agent which can perform a packet encapsulation/decapsulation function. A packet whose destination address is a mobile host is intercepted by a packet forwarding agent. Intercepted packets are encapsulated and forwarded using the address of another agent that is located close to the mobile host. The destination agent strips the encapsulation header and relays the original packet to the mobile host.

The major problem with this approach is that routing is always sub-optimal, unless the packet forwarding agent is located close to the source. Another problem is the fragmentation and reassembly required at the packet forwarding agent. It has been observed that the majority of packets in the network are either very short or very large in size. Packet fragmentation may occur at the packet forwarding agent if the extra bytes added for encapsulation cause a packet to exceed its maximum transmission size. This may also happen at an MR when it inserts the LSR option. However, the risk of fragmentation is higher in an encapsulation based approach because the size of an encapsulated packet is larger than a packet with LSR option.

The LSR approach also has a few shortcomings. The first problem is that UDP[8] agents do not perform route reversal. As a result, UDP packets originating from a stationary host are always routed sub-optimally via a mobile router. Notice that if there is no encapsulating agent close to the stationary host, any scheme based on encapsulation approach will also suffer from this problem. One approach to get around this problem is to distribute the MR functionality among multiple cooperating MR agents and place them close to the UDP sources. This also improves the robustness of the system.

Another problem with the LSR option is that there is a potential security risk associated with its use. The authentication mechanisms which are based on the source address lookup can easily be broken with the use of the LSR option. For this reason, many routers disable this. Note that encapsulation has exactly same problem unless each MSR [3] knows every other MSR. Although this is a genuine problem, we should not overlook the benefits of source routing only on the basis of this limitation. IP itself has a lot of security holes. Besides, mobile networks, due to wireless links, are quite prone to security attacks. Better methods for data and address encryption are required to make them tolerant against impersonation attempts. In future, some of those techniques can be used to make the LSR usage more secure. Another concern that has been raised is related to the performance aspect of LSR. Compared to normal IP packets, more processing time is required to process packets which contain options. Existing routers manufactured by router vendors have specialized code to rapidly forward IP packets that do not contain any IP options. It is, however, not difficult to optimize this, since a similar performance enhancement could be made for

packets containing precisely the IP LSR option with option data of the expected length(s). In the normal case, routers would not have to do any further processing, but would only have to forward the IP packet unchanged.

The following is a quick summary of the comparison of the two approaches.

Criteria	LSR	Encapsulation
Existing Implementation	Usually Poor	Nonexistent
Impersonation	Easy spoof	Easy spoof
Router Efficiency	Not optimized	Optimized
Additional Fragmentation	Less	More
ICMP error propagation	Correct	Incorrect
Fragmentation/Reassembly	End-to-End	Done by encapsulating agent

In conclusion, we have shown that LSR provides a fast and elegant solution to the problem of providing seamless networking for mobile hosts using the Internet Protocol. The use of LSR also enables optimal routing in most cases, without incurring the expense of agents interceding for the destination hosts, as long as those hosts implement IP as it has been specified now for a decade. The additional protocol requirements are minimal, and with a few changes to our system could be eliminated almost entirely.

Besides using LSR our approach demonstrates a modular construction with good separation between layer 2 (MAC/link layer) and layer 3 (IP) functions. The beaconing, cell discovery, and cell switch mechanisms are all quite distinct from location update and routing mechanisms. Either feature could be replaced without substantially affecting the other. Indeed, new cell discovery mechanisms are likely to become commercially available at protocol layer 2, and our system will still work in almost exactly the same way.

Our construction also eliminates any addressing relationship between the MAS and the MH. The MH needs to discover the wired IP address of the MAS, but that IP address need have no relationship to its own IP address. The MH does not ever have to know anything about the MAS's wireless IP address – if indeed the MAS even has a wireless IP address. The MAS is quite passive in our system, and there is no requirement for intercommunication between any two MASs. This will be very helpful to those intending to design secure wireless systems, insofar as MASs become more like dumb transceivers and do not have to be trusted agents.

Lastly, our system has shown itself to be quite practical. We have had test systems running in our laboratory for almost a year with the current implementation (previous systems ran mostly as a simulation). The mobile hosts are quite usable, and all new kernels and other programs have been installed by wireless file transfers of various sorts. The application transparency has been just about perfect; X11, NFS, and all the normal networking software has required absolutely no change. The only change above the IP network layer has been to eliminate the route-caching feature built into TCP for performance reasons. We have not noticed any significant slowdown in operation of the wireless network except for bulk data transfers, and that is due to the relatively slow speed of our wireless adapter hardware.

8 Acknowledgements

We wish to gratefully acknowledge Yakov Rekhter for his contribution to the design, implementation, and debugging of the system described in this paper. Also we would like to express thanks to Dheeraj Sanghi and Debanjan Saha for reviewing the paper.

References

[1] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, Oct. 1989.

[2] Douglas Comer. *Internetworking with TCP/IP.* Prentice Hall Mcgraw Hill, 1988.

[3] John Ioannidis, Dan Duchamp, and Gerald Q. Maguire Jr. IP-based Protocols for Mobile Internetworking. In *Proceedings of ACM SIGCOMM*, pages 235–245, 1991.

[4] John Ioannidis and Gerald Q. Maguire Jr. The Design and Implementation of a Mobile Internetworking Architecture. In *Proceedings of Winter USENIX*, pages 491–502, San Diego, CA, Jan 1993.

[5] Charles Perkins. Providing Continuous Network Access to Mobile Hosts Using TCP/IP. In *Joint European Networking Conference*, May 1993.

[6] Charles Perkins and Yakov Rekhter. Short-cut Routing for Mobile Hosts. Internet draft, July 1992.

[7] Charles Perkins and Yakov Rekhter. Support for Mobility with Connectionless Network Layer Protocols. Internet draft, November 1992.

[8] J. Postel. User Datagram Protocol. RFC 768, Aug 1980.

[9] J. Postel. Internet Control Message Protocol. RFC 792, Sep 1981.

[10] J. Postel. Internet Protocol. RFC 791, Sep 1981.

[11] J. Postel. Transmission Control Protocol. RFC 793, Sep 1981.

[12] Fumio Teraoka and Mario Tokoro. Host Migration Transparency in IP Networks. *Computer Communication Review*, pages 45–65, Jan 1993.

[13] Fumio Teraoka, Yasuhiko Yokote, and Mario Tokoro. A Network Architecture Providing Host Migration Transparency. In *Proceeding of ACM SIGCOMM*, Sept 1991.

[14] Hiromi Wada, Takashi Yozawa, Tatsuya Ohnishi, and Yasunori Tanaka. Mobile Computing Environment Based on Internet Packet Forwarding. In *proceeding of Winter USENIX*, pages 503–517, San Diego, CA, Jan 1993.

Providing Connection-Oriented Network Services to Mobile Hosts[1]

Kimberly Keeton, Bruce A. Mah, Srinivasan Seshan, Randy H. Katz, Domenico Ferrari
{kkeeton,bmah,ss,randy,ferrari}@CS.Berkeley.EDU
Computer Science Division
University of California at Berkeley

Abstract

Mobile computers using wireless networks, along with multimedia applications, are two emerging trends in computer systems. This new mobile multimedia computing environment presents many challenges, due to the requirements of multimedia applications and the mobile nature of hosts. We present several alternative schemes for maintaining network connections used to provide multimedia service, as hosts move through a nano-cellular radio network. These algorithms modify existing connections by partially re-establishing them to perform handoffs. Using a simple analytical model, we compare the schemes on the basis of the service disruption caused by handoffs, required buffering, and excess resources required to perform the handoffs.

1.0 Introduction

Two technological trends of the 1990s are the emerging use of wireless computers and network support for multimedia services. The products of these trends hold forth the promise of being combined in innovative ways to provide applications such as mobile digital video and audio conferencing. The new computing environment presented by wireless multimedia personal communication systems such as discussed in [Sheng92] introduces many challenges, because of the requirements of multimedia applications and the mobile nature of the hosts.

Multimedia applications typically have strict requirements such as delay, delay jitter, throughput, and reliability bounds. Real-time network services are designed to guarantee these performance parameters to applications that request them. In order to provide these performance guarantees to individual conversations, approaches such as [Ferrari92] rely on connection-oriented networks and resource reservation. Per-connection resource allocation in a connection-oriented network is necessary to ensure that guarantees will not be violated under conditions of heavy network congestion. (In connectionless network environments, congestion typically causes decreased throughput, increased delay, and even an increased probability of packet loss.) These methods, which are designed for wired networks, do not directly address the mobility of hosts. While these performance guarantees are necessary for the delivery of multimedia data to mobile hosts, our concern in this paper is not providing these guarantees, but instead maintaining the connections in use as a host moves within the network.

User mobility forces networks to cope with new dynamics of routing and resource location. Generally, mobile networks are composed of a wired, packet-switching, backbone network and a wireless (e.g. cellular radio or infrared) network. The wireless network is organized into geographically-defined cells, with a

1. This research was supported by the Department of Energy under grant DE-FG03-92ER25135/A000, by the National Science Foundation and the Defense Advanced Research Projects Agency (DARPA) under Cooperative Agreement NCR-8919038 and by the Corporation for National Research Initiatives, AT&T Bell Laboratories, Hitachi Ltd., Hitachi America Ltd., Pacific Bell, the University of California under a MICRO grant, and the International Computer Science Institute. Kimberly Keeton was supported by an AT&T Graduate Fellowship. Bruce A. Mah was supported by an NSF Graduate Fellowship.

This paper, as well as other works on real-time computer networks is available for anonymous FTP from `tenet.ICSI.Berkeley.EDU`.

control point called a *base station* (BS) in each of the cells. The base stations, which are attached to the wired network, provide a gateway for communication between the wireless network and the backbone interconnect. As a *mobile host* (MH) travels between wireless cells, the task of forwarding data between the wired network and the mobile host must be transferred to the new cell's BS.

This process, known as a *handoff*, must maintain end-to-end connectivity in the dynamically reconfigured network topology. Since our model of network communication is connection-oriented, each of the MH's connections (or channels) and the associated connection state must somehow be transferred to this new BS. The effectiveness of the rerouting performed by the handoff algorithm is determined by several criteria. In particular, it is desirable to minimize the service disruptions (such as loss of motion due to the replay of the last frame when subsequent frames do not arrive on time) and overheads such as latency, MH and BS buffering, and excess reservation of network resources.

Prior work related to host mobility does not simultaneously address the issues of connection-oriented service and de-centralized control. Much has been accomplished in the context of providing support for the Internet protocol suite in a wireless environment [Ioannidis91, Teraoka91]. However, because its network layer protocol (IP) is connectionless, network performance cannot be guaranteed under conditions of high load. Conversely, the cellular telephone network uses circuit switching to provide connection-oriented services. It maintains these connections during handoffs through highly centralized knowledge and control by the mobile telephone switching offices.

Two straightforward solutions to the challenge of connection-oriented handoffs are forwarding data from the original BS to the new BS and establishing a new connection between the multimedia source and the new BS. Because these approaches incur considerable overheads, we propose several alternate algorithms that modify the existing connections by partially re-establishing them to perform handoffs. The first protocol capitalizes on the logical locality of geographically adjacent cells to partially re-establish the connections during handoff. The second uses multicast facilities to provide connectivity to the new base station when the host moves.

We present a quantitative comparison of these strategies using analytically derived formulas for connection setup latency, required buffering, and excess resource reservation. We compare the algorithms and evaluate their feasibility, given the physical limitations imposed by current wireless network technology. Our basis for comparison is the case where connections are fully re-established to the source.

2.0 Environment

The physical and logical aspects of the networking environment will play a large role in determining the nature of the handoff algorithms and their requirements.

2.1 Computing Environment

The technologies used in the mobile system have a significant impact on the effectiveness of the handoff schemes. We have chosen several technology points, based on typical portable computers and switch-based networks, to characterize our computing environment. For instance, most mobile computers have fully functional CPUs, although the low-power requirements of the mobile host may limit its computational capabilities. We assume, however, that the mobile hosts contain enough processing power to run applications and the necessary communication protocols.

The wireless and wired network technologies available today define various aspects of mobile communication. We assume a nano-cellular radio network (with cells less than ten meters in diameter) covering the interior of a building. Users (and hence mobile hosts) move occasionally and slowly compared to the speed of the network. Mobile hosts communicate through the radio network to a subset of the base stations; each MH can communicate with at most one BS at a time. These base stations are gateways between the radio network and a switch-based store-and-forward backbone network, and facilitate communication between the mobile hosts and multimedia servers on the backbone network. Base stations are single-homed; that is, they have

only one physical connection to the wired network. For simplicity, hosts and switches are interconnected in a hierarchical topology resembling a tree. The switches support duplex connections. (For convenience we refer to the direction towards the MH as the *downlink* and the direction away from the MH as the *uplink*.) Finally we assume that the switches and BSs can be programmed to support our handoff algorithms.

The characteristics of Code Division Multiple Access (CMDA) radio (< 2 GHz carrier frequency) nano-cellular networks [Schilling91] define some of the wireless communication patterns. In a radio network, there is generally considerable overlap between the cells of the network. (As stated before, however, a MH can communicate with only one BS at a time.) Each host in a radio network knows which BSs are "in range" and which of those BSs has the strongest radio signal. The relative strength of the radio signal from the current BS and the next strongest radio source gives the MH an indication of when it is close to a new cell. The MH can also use the CDMA codes to determine which cell it is approaching.

However, handoff algorithms cannot rely with perfect certainty on the availability of this information. Radio networks may have *null regions*, areas in which a MH loses wireless contact with its BS, possibly due to interference with radio propagation. The cell overlap information cannot be completely relied upon because it may be possible to enter a null region in one cell and exit the region in a different cell. Some radio networks may also have sharp boundaries between some cells (for example, doorways). Transitions across these boundaries may occur too quickly for any advance warning to be useful. Due to these two characteristics of wireless networks, our approach uses information about an impending cell transition as a hint to improve performance, rather than as an integral component of the handoff algorithms.

2.2 Communication Paradigm

We assume that the MHs will provide various multimedia services, as described in [Sheng92]. Many multimedia services, such as audio-video conferencing or video playback, have associated with them performance requirements that must be met to guarantee acceptable service to the users. [Ferrari90] describes the requirements that some typical applications place on networks. The Tenet Real-Time Protocol Suite [Ferrari92] is one approach to providing these *real-time performance guarantees* in packet-switching networks; this approach relies on resource reservation in a connection-oriented network environment.

A connection-oriented network paradigm with per-conversation resource reservation is needed to provide performance guarantees. In a connectionless scheme, successive packets may follow different paths in the network, and hence incur different delays in reaching the destination. Packet delays in this environment are much harder to control than if all packets followed the same path, as in a connection-oriented scheme. Per-conversation resource reservation is necessary to ensure that sufficient resources are available for real-time traffic at all times, including periods of high network load and congestion. Because network resources are finite, an admission control policy must be administered, to ensure that resources are not over-subscribed.

Before communication can take place, a real-time channel (a connection with performance guarantees) must be established. Channel establishment involves a single source-routed round-trip pass of control messages from the source to the destination of the connection and back.[2] On the forward pass, admission control tests are performed to ensure that the network will be able to satisfy this channel's performance requirements without violating the guarantees of existing real-time channels. Assuming the tests succeed, node resources (such as bandwidth) are tentatively allocated, pending the acceptance of this channel at all other participating nodes. On the return pass of a successful establishment, the resources of each node are committed to the channel. [Mah93] describes in detail a mechanism and protocol by which real-time channels can be established and managed.

We also assume that all connections have at most one MH as an endpoint (e.g., at least one endpoint is a fixed host on the wired network). This assumption somewhat simplifies the design of the algorithms, as they do not need to consider the case where both ends of a connection move simultaneously.

2. The Tenet real-time channels are simplex unicast, but it is a fairly simple matter to support duplex real-time channels. Ongoing work is aimed at providing real-time multicast network services.

2.3 Topological Knowledge

The algorithms that we present modify existing connections to perform handoffs. We call the point where a connection is modified the *crossover point*. Choosing this crossover point requires some knowledge of the network topology. We assume, however, that control entities have only local knowledge of the network (such as next-hop routing information); this allows them to function without needing large status updates from their peers. This assumption also prevents a single entity from making rerouting decisions independently. Therefore, multiple control entities must collaborate using a distributed algorithm to determine the crossover point.

3.0 Algorithms

In this section we discuss three different schemes for handoffs in connection-oriented mobile networks. We present a Full Re-Establishment scheme as a basis for comparison. We then introduce two new algorithms, Incremental Re-Establishment and Multicast-Based Re-Establishment. The algorithms all rely on a number of pre-existing connections in the network, to be used primarily for control purposes. We assume that any two network nodes connected by a link have a control channel between them. Every pair of base stations responsible for physically adjacent cells also has a control connection between them. (We note that these base stations may not be directly connected by a physical link.) In addition, these algorithms require the BSs to buffer a bounded amount of data that has already been transmitted to the mobile host, to ensure that no data is lost when the mobile host moves to an adjacent cell. The algorithms make no attempt to re-order out-of-order packets. They will, however, attempt to prevent data from becoming out of order as the result of a handoff operation.

For each algorithm, we give a short description, followed by a brief example of its application re-rerouting a single channel. We also present the optimized case which takes advantage of the cell overlap hint. In each of the examples, the MH moves from the cell whose point of control is BS 1 into the cell corresponding to BS 2. Thus, in all cases, BS 1 is considered to be the "old BS" and BS 2 is considered the "new BS". Numbered lines correspond to control messages exchanged between network entities to complete the handoff.

3.1 Full Re-Establishment

The Full Re-Establishment (FR) algorithm executes handoffs by establishing a set of completely new channels between the MH and the servers.

3.1.1 Full Re-Establishment without Hints

The "Without Hints" part of Figure 1 depicts the communication necessary to perform the FR scheme in the general case where the MH has no advanced warning that a handoff from BS 1 to BS 2 is about to occur. Once the MH enters the new cell, it identifies itself to the new cell's BS, BS 2, and requests that its connections be rerouted through the new BS (message labeled "1" in the diagram). Included in this greeting message is an identifier for the old BS, BS 1, as well as a list of the identifiers for the various connections originating or terminating on the MH. After BS 2 acknowledges this greeting with message (2), data transmission from the MH may begin.

In order to avoid an interruption in data service, BS 2 immediately uses the pre-existing BS-to-BS control channel to request that downlink data from each of the MH's connections be forwarded from the original BS (3). Message (3) also requests that BS 2 be allowed to forward uplink data from the MH through BS 1 to the Server. BS 1 acknowledges these requests, and may begin forwarding data to BS 2 just after (4). Once (4) has been received, BS 2 may begin forwarding data transmitted by the MH through BS 1.

While this data forwarding is occurring, BS 2 begins establishing connections to each of the appropriate Servers on the network (5). Once a connection is fully established (6), the new BS informs the MH (7) and sends a message (8) to the Server requesting that it redirect all data transmissions down the newly estab-

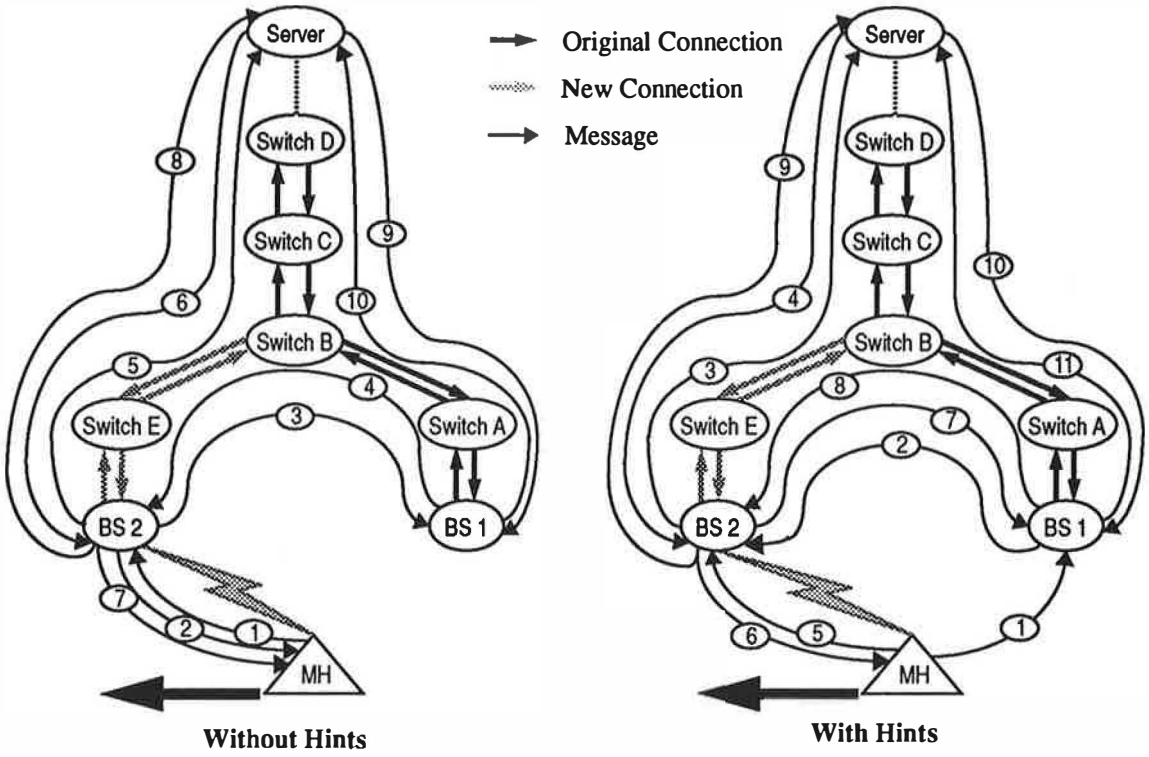


FIGURE 1. Full Re-Establishment

lished downlink. The downlink portion of the connection between Switch B and BS 1 is then torn down in step (9). The redirected downlink data, and the remainder of the data forwarded through BS 1 to the new BS, is buffered at BS 2 to ensure in-order delivery to the MH.

Once the new portion of the connection has been established, data from the MH can flow directly to the Server over the new connection. To preserve in-order delivery of the data from the MH to the Server, this uplink data must be initially buffered at BS 2 until the uplink data forwarded through BS 1 is drained from the old portion of the connection. This buffering is proportional to the difference between the transmission delays along the new path and the forwarding/old path. Once all messages forwarded through BS 1 have been delivered to the Server, the uplink portion of the old connection is torn down (10) and uplink data may flow directly through BS 2. The handoff is completed once all of the new connections have been created and the old connections destroyed.

3.1.2 Full Re-Establishment with Hints

In a radio network, the MH can take advantage of the overlap between adjacent cells to detect that it is entering a new cell before it loses contact with its current BS. This "hint" allows the MH to request that the current BS establish in advance new connections between the Servers and the new BS. This scenario is shown in the "With Hints" portion of Figure 1. At the hint time, the MH requests (1) that its current BS (BS 1) send a list (2) of active connections to the new BS (BS 2). The new BS may then establish each of the connections to the appropriate network Server (3). A successful establishment is indicated to BS 2 by the acknowledgment in step (4).

During this pre-establishment, the MH ceases communications with the old BS and begins communicating with the new BS (5). As in the more general algorithm, this greeting is then acknowledged (6). Depending on how much time has elapsed, the new connections may or may not have been established to the new BS. In the best case, establishment for each of the new connections has been completed, and message (6) also notifies the MH of which connections have been established. As in the more general algorithm without hints, BS 2 then initiates a forwarding request for all of the data transmitted during the radio communication switcho-

ver period on a given connection from the Server to BS 1 (7, 8). Downlink data can then be forwarded across the pre-existing channel by BS 1. However, the uplink data is not forwarded, but sent only on the newly established connection to the Server. To ensure in-order delivery of the data from the MH to the Server, this uplink data must be initially buffered at BS 2 until all of the data that was transmitted by the MH (while in the old cell) can be sent along the old connection. This buffering is proportional to the difference between the transmission delays along the two paths, taking into consideration the time for the MH to make contact with the new BS.

While downlink data is being forwarded, the new BS sends a message (9) to the server requesting that data be rerouted to the new connection. Once the Server switches active connections, it begins deleting the channel to the old BS (10). After all uplink data has been transported to the Server, the old connection is completely torn down (11).

If the MH arrives at the new BS before its connections have been established (but after the hint processing has begun), forwarding for both the down and uplink is initiated as in the general case with no hints. The MH is then notified as each connection is established. At this point, the new BS sends a message to the Server indicating that it may begin sending downlink data over the newly established channel. The teardown of the old channel proceeds as in the general case with no hints.

If the MH does not end up fully entering the cell it was approaching but instead eventually moves out of its range, it must send another message to its original BS revoking the hint. This results in a tear-down of the connections that had been set up in anticipation of the arrival of the MH.

3.2 Incremental Re-Establishment

In contrast to the FR algorithm, the Incremental Re-Establishment (IR) scheme attempts to re-use as much of an existing connection as possible, creating only the portion between the crossover point and the new cell's BS. The corresponding portion of the original connection is then torn down.

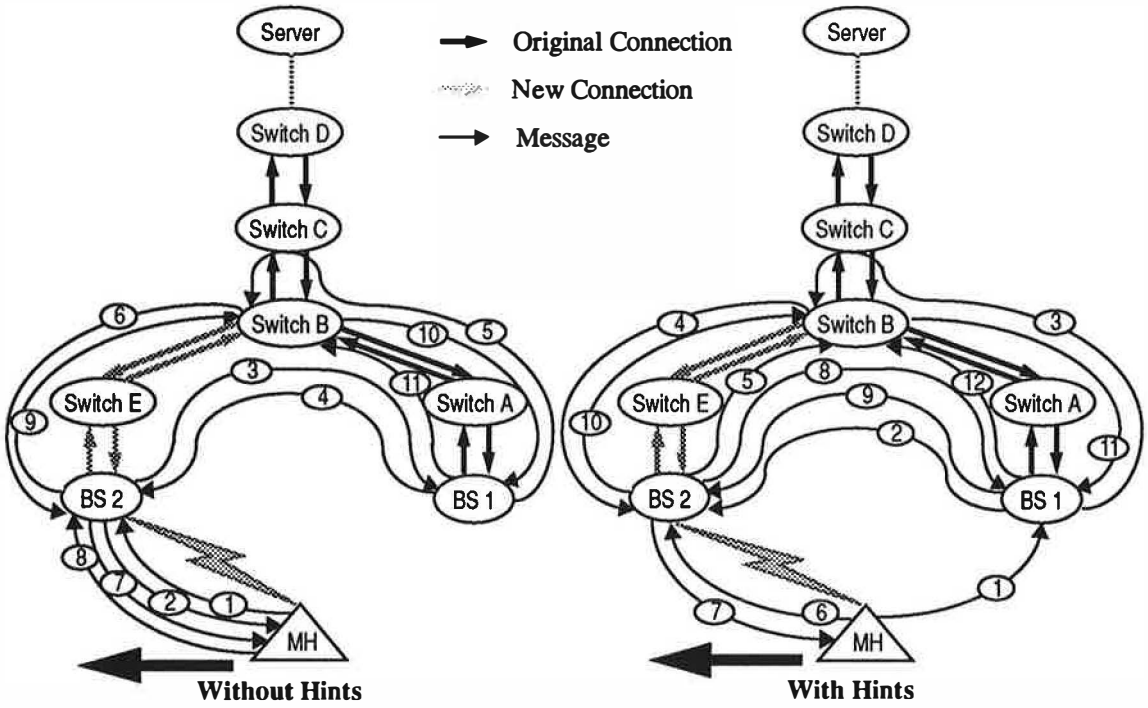


FIGURE 2. Incremental Re-Establishment

3.2.1 Incremental Re-Establishment without Hints

The "Without Hints" portion of Figure 2 illustrates the application of the IR scheme in the general case where the MH has no advanced warning of the impending handoff. When the MH first arrives in a new cell, it sends a greeting message (1) to the new BS, BS 2, which contains a list of connections to be rerouted as well as the identity of the old BS. Data transmission from the MH to BS 2 begins after BS 2 acknowledges this greeting (2). As in the FR scheme, BS 2 requests that BS 1 forward the downlink and uplink data from each of the MH's connections across the BS-to-BS connection (3, 4).

In order to reuse a portion of the existing connection, message (3) also implicitly requests on behalf of the new BS that the old BS invoke the distributed crossover point location process. This location algorithm is initiated by the old BS because the new BS has no knowledge of the connection path to the old BS and possesses only local topological knowledge. BS 1 invokes this decision process by backtracking along the original connection route one hop at a time (5). Each switch along this route decides whether it knows the appropriate crossover point by examining its routing tables. If the switch uses different ports to reach the old and new BSs, it continues to forward this reroute message (Switches A and B in step 5). If the switch uses the same port to reach both the old and new BSs (Switch C), it knows that the switch one hop in the downlink direction (Switch B) is the crossover point. Switch C then communicates this fact to Switch B.

Once the crossover point has been identified, the new portion of the connection must be established and the old portion torn down. The first step, establishing the new partial connection between the crossover point (Switch B) and the new BS (BS 2), is performed as it would be in a wired network (6). This connection is established to the MH using message (7). Assuming establishment is successful, acknowledgments are sent from the MH to BS 2 (8) and from BS 2 back along the path to Switch B (9).

Message (9) is also an implicit request for Switch B to redirect all data transmitted by the Server down the newly established downlink. As in the FR algorithm, buffering at BS 2 is used to allow in-order delivery of downlink data to the MH. The downlink portion of the connection between Switch B and BS 1 is then torn down in step (10). After step (9), data from the MH can flow directly to the Server through BS 2. This uplink data is buffered at BS 2 to allow the uplink data forwarded through BS 1 to be delivered to the crossover point. Once all messages forwarded through BS 1 to the Server have been delivered to Switch B, the uplink portion of the old connection is torn down (11) and uplink data may flow over the new connection. When all new connections have been created and old connections destroyed, the handoff is complete.

3.2.2 Incremental Re-Establishment with Hints

The "With Hints" portion of Figure 2 shows how the cell overlap information can be used to facilitate the handoff. The MH informs the current BS, BS 1, of the potential new BS, BS 2, and requests that new partial connections be established through the appropriate crossover points to the new BS (1). BS 1 communicates the list of connections about to be re-established to the new BS (2), and then invokes the crossover location algorithm (3). Once located, the crossover point, Switch B, begins to establish a new connection to the new BS (4). A successful establishment is indicated to Switch B by the acknowledgment in step (5).

While the connection to the new BS is being pre-established, the MH moves completely into the new cell, where it identifies itself (as in the more general algorithm) to BS 2 (6) and is acknowledged (7). Analogous to the FR scheme with hints, in the optimal case where all connections are established, message (7) contains identifiers for the already-established connections. Similarly, BS 2 initiates downlink data forwarding from BS 1 (8, 9) and then requests that the crossover point redirect downlink data over the new connection (10). Again, uplink data is not forwarded through the old BS. Buffering to ensure in-order delivery of both downlink and uplink data is performed as in the FR scheme with hints, except that the endpoint of the two paths under consideration is the crossover point rather than the server. Finally, suboptimal cases are handled in a manner analogous to that of the FR scheme with hints.

3.3 Multicast-Based Re-Establishment

To support video conferencing and other distributed applications, some networks support multicast connections with dynamically changing memberships. The use of multicasting has several interesting ramifications. Because data for the downlinks are transmitted simultaneously to multiple base stations during the interim, the actual switchover can be fairly quick, with decreased buffering. This scheme also has an advantage in that only a small layer of functionality needs to be added on top of the multicast facility in order to support mobility.

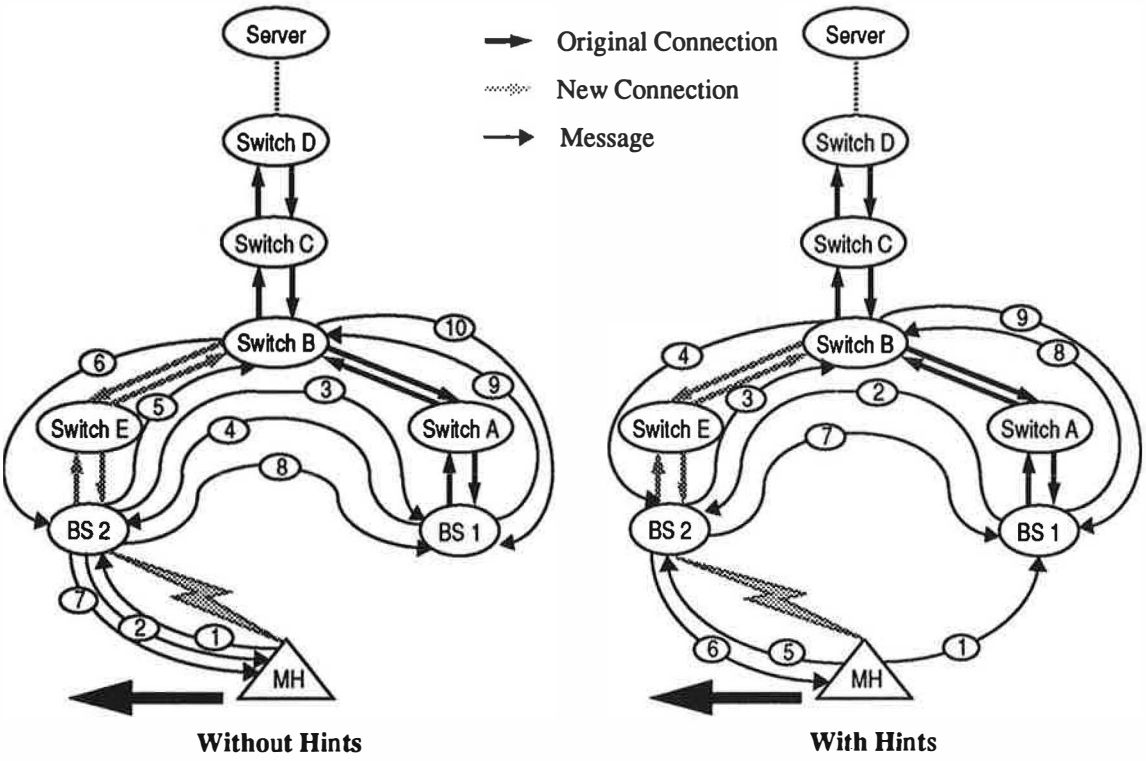


FIGURE 3. Multicast-Based Re-Establishment

3.3.1 Multicast-Based Re-Establishment without Hints

The "Without Hints" diagram in Figure 3 illustrates the operation of the MB handoff algorithm. The MH detects that it is entering a new cell, so it acquires a wireless channel to the new BS, BS 2, and sends it a greeting message (1). This message contains an identifier for the old BS, BS 1, as well as a list of the identifiers for the various multicast channels originating or terminating on the MH. This greeting is immediately acknowledged by BS 2 (2). BS 2 then sends this channel list to the old BS along the pre-existing BS-to-BS connection (3), requesting that all data for each channel be forwarded to the new BS. In addition, BS 2 requests that it be allowed to forward data from the MH through BS 1. Upon receipt of an acknowledgment (4), BS 2 begins transferring forwarded data from BS 1 to the MH and forwarding MH data destined for the server.

Concurrently with the forwarding requests, BS 2 executes a multicast join operation (which establishes a new branch connecting BS 2 with the existing channel) for each existing channel to add itself to the multicast channel (5). Upon receiving an acknowledgment indicating a successful establishment (6), the new BS notifies the MH of the successful join operation (7). BS 2 synchronizes the data arriving down the new branch with the data being drained from the old BS. To preserve in-order delivery of the data from the MH to the server, this data must be initially buffered at BS 2 until the data forwarded through BS 1 is drained from the old portion of the connection.

When all of the multicast joins have completed and all necessary data have been obtained from the old BS, the new BS sends a completion message to the old BS (8) over the control connection. At this point, BS 1 can assume that it has no more responsibilities to the MH, and so it executes a multicast leave operation for each of the channels associated with the MH (9, 10). This operation deallocates node resources along the path from the BS to the first node in common with another branch of the multicast channel. The handoff is complete after the last branch has been torn down.

3.3.2 Multicast-Based Re-Establishment with Hints

The "With Hints" diagram in Figure 3 illustrates the advance setup the network performs in response to the hint and a best case scenario for an ensuing handoff. The hint is delivered as a message to BS 1 (1) identifying the potential new BS, BS 2. BS 1 then notifies BS 2 (2) that it should initiate multicast join operations to all of the MH's channels in anticipation of a handoff (3, 4). When the MH loses contact with BS 1 and enters BS 2's cell, it identifies itself with BS 2 as before (5). Due to the hint, BS 2 will have already initiated joins for all of the channels associated with the MH. When the joins have been successfully completed, BS 2 notifies the MH (6) and drains any data buffered for the MH. Finally to complete the handoff, BS 2 sends a completion message to BS 1 (7), which then executes a multicast leave operation for each MH channel (8, 9). Buffering requirements on the BSs are similar to the FR and IR algorithms with hints.

4.0 Analysis

In this section, we describe our analysis of the algorithms. We use an analytical model of the algorithms and a characterization of the network to compute the values of various metrics describing the overheads involved in connection rerouting.

4.1 Parameters and Metrics

We use a set of technology-dependent parameters, shown in Table 1, to characterize the network. We assume no contention for link bandwidth or other network resources. Third generation wireless networks are capable of supporting megabit per second speeds, and current experimental LANs support gigabit per second bandwidths. Protocol processing time is assumed to be constant except in the case where admission control tests need to be performed, as Tenet-style admission control tests are processing intensive (e.g., 25 ms on a RISC workstation like a DEC 5000/240).[3] We have placed an upper bound of 50 Bytes on control messages, as they generally contain a small, fixed amount of information. (Channel establishment messages may be much larger, however.) The maximum data packet size is chosen to be 8 KBytes, which is an average packet size in many typical local area networks. Finally, the time taken to acquire a wireless channel is dependent on the actual wireless network system in use.

Symbol	Definition	Value
BW_{wl}	Bandwidth of the wireless link.	1 Mbps [Schroeder92]
BW_{w}	Bandwidth of the wired backbone network.	1 Gbps [Brodersen93]
L_{wl}	Latency of the wireless link, including data link and network layer processing.	7 ms [Brodersen93]
L_{w}	Latency of a link in the wired backbone, including data link and network layer processing.	500 µs [Zhang92]

TABLE 1. Technology-Dependent Network Parameters

3. The values for protocol processing times were derived from measurements of the Tenet Real-Time Channel Administration Protocol (RCAP) running on DECstation 5000/240 workstations.

Symbol	Definition	Value
PPT_{fixed}	Protocol processing time for control messages.	3 ms
PPT_{adm}	Protocol processing time for steps where admission control is to be performed, in excess of fixed protocol processing time.	25 ms
S_{ctrl}	Upper bound on the size of a control message.	50 bytes
S_{data}	Maximum size of a data packet.	8192 bytes
$T_{acquire}$	Time for a MH to acquire a wireless channel to a base station.	20 ms [Brodersen93]

TABLE 1. Technology-Dependent Network Parameters

Each network connection to and from a mobile host at the time of handoff can be characterized by the set of parameters listed in Table 2. These parameters are dependent on the locations of the mobile host and its multimedia servers, as well as the topology of the network. By varying these parameters, we can examine the overheads associated with each rerouting algorithm over varying lengths of connections.

Symbol	Definition
H_{new}	Number of hops to the crossover point from the new (destination) BS.
H_{old}	Number of hops to the crossover point from the old (original) BS.
H_{ctrl}	Number of hops between the old and new BSs along their control channel.
H_{server}	Number of hops between a MH and its multimedia server.

TABLE 2. Parameters Characterizing Network Connections.

We can measure the performance of the various handoff algorithms using the metrics listed in Table 3. The values of these metrics, evaluated for a given set of technology-dependent parameters and various parameters of network connections, will give some idea of the effectiveness of the different schemes. $T_{disrupt}$ corresponds to the service disruption time, that is, the time during which the MH cannot receive data on its downlink. (This disruption may manifest itself as a pause in the playback of video.) B_{mh}, $B_{bs,down}$, and $B_{bs,up}$ correspond to the buffering required on the MH and the BS to prevent data from becoming out of order as a result of a handoff operation. The bandwidth-space-time products are one measure of the added network resources required to do the handoff. They are essentially the product of the bandwidth provided to a connection, multiplied by the length of the connection and the amount of time that the connection is in existence. P_{excess} measures the network resources that are allocated to inactive connections (for example, a channel or portion of a channel that has been established but is not being used to carry data). P_{fwd} is a measure of the network resources reserved for the forwarding of packets between adjacent BSs during handoff.

Symbol	Definition
$T_{disrupt}$	Service disruption time, the time during which the MH cannot receive data on its downlink.
B_{mh}	Buffering required on the MH for buffering of uplink data during handoff.
$B_{bs,down}$	Buffering required in the BSs for buffering of downlink data during handoff.
$B_{bs,up}$	Buffering required in the new BS for buffering of uplink data during handoff.
P_{excess}	Excess bandwidth-space-time product used by inactive channels during handoff.
P_{fwd}	Bandwidth-space-time product used by data being forwarded during handoff.

TABLE 3. Metrics for Comparing Handoff Algorithms

4.2 Derivation of Metrics

We now present the derivation of the metrics listed in Table 3 for the Incremental Re-Establishment algorithm, in the case that there are no hints regarding advance warning of a handoff. The metrics for the other handoff algorithms can be derived in a similar way. Due to space limitations, these derivations are not presented here.

This analysis assumes perfect delivery of control messages. (An analysis of the general case in which control messages can be lost during the handoff is an exercise in protocol verification which we intend to address in the future.) We also assume that the maximum throughput for a connection is the throughput of the bandwidth of the wireless link. All of our computations for buffering requirements are derived on a per-channel basis, with the worst case being that in which the channel's throughput is equal to the wireless link's total bandwidth.

We compute the time taken for each of the scheme's control messages to be transmitted, forwarded, and if necessary, processed. Message numbers for this section are those from Section 3.2.1 and Figure 2. Message (1) is a greeting from the MH to the BS in the new cell. The latency needed for this message is the sum of the time to acquire a wireless channel, the transmission time of a control message on that channel, the propagation time on the wireless link, and the fixed protocol processing time on the new BS.

$$T_1 = T_{acquire} + \left(\frac{S_{ctrl}}{BW_{wl}}\right) + L_{wl} + PPT_{fixed} \tag{1}$$

Message (2) is an acknowledgment of the greeting back to the mobile host; its total delay is simply the sum of the transmission and propagation times, plus processing time in the mobile host.

$$T_2 = \left(\frac{S_{ctrl}}{BW_{wl}}\right) + L_{wl} + PPT_{fixed} \tag{2}$$

Immediately after sending Message (2), the new BS sends a request (3) for forwarding of both uplink and downlink data to the old BS, using the control channel between the two (physically) adjacent base stations. Its total latency is the end-to-end delay through the control channel plus the fixed protocol processing time in the old BS.

$$T_3 = \left[\left(\frac{S_{ctrl}}{BW_w}\right) + L_w\right] H_{ctrl} + PPT_{fixed} \tag{3}$$

The next message, Message (4), is an acknowledgment of the forwarding request, sent from the old BS to the new BS. Its total latency is computed similarly to that of Message (3).

$$T_4 = T_3 = \left[\left(\frac{S_{ctrl}}{BW_w}\right) + L_w\right] H_{ctrl} + PPT_{fixed} \tag{4}$$

Immediately after sending Message (4), the old BS begins the distributed crossover point location process (5). The messages needed to find the crossover point will incur a latency (T_5) equal to the transmission and propagation time along each hop, plus the fixed control protocol processing time in each switch along the path from the old BS to the crossover point. We note that the PPT_{fixed} term in each hop's latency is necessary because, in contrast to Messages (2) and (3), the switch controller at each hop needs to do some processing of the control message (specifically, to determine whether it knows the crossover point).

$$T_5 = \left[\left(\frac{S_{ctrl}}{BW_w}\right) + L_w + PPT_{fixed}\right] H_{old} \tag{5}$$

Message (6) is a partial channel establishment from the crossover point to the new BS. Admission control tests need to be executed at each switch along this path, which implies a delay of PPT_{adm} at each hop in addition to the normal PPT_{fixed} required for normal control message processing.

$$T_6 = \left[\left(\frac{S_{ctrl}}{BW_w}\right) + Lw + PPT_{fixed} + PPT_{adm}\right] H_{new} + PPT_{adm} \quad (6)$$

Message (7) completes the partial channel establishment from the new BS to the mobile host; therefore the bandwidth and link latency are those of those of the wireless link.

$$T_7 = \left(\frac{S_{ctrl}}{BW_{wl}}\right) + L_{wl} + PPT_{fixed} + PPT_{adm} \quad (7)$$

Message (8) is the acknowledgment of the partial channel establishment across the wireless link.

$$T_8 = T_2 = \left(\frac{S_{ctrl}}{BW_{wl}}\right) + L_{wl} + PPT_{fixed} \quad (8)$$

Acknowledgment of the partial channel establishment (back to the crossover point) is completed by Message (9). This control message retraces the path of the partial establishment, hop by hop.

$$T_9 = \left[\left(\frac{S_{ctrl}}{BW_w}\right) + L_w + PPT_{fixed}\right] H_{new} \quad (9)$$

Message (10) begins the teardown of the partial connection from the crossover point to the old BS. As with the Message (5), it must travel hop by hop between switch controllers.

$$T_{10} = T_5 = \left[\left(\frac{S_{ctrl}}{BW_w}\right) + L_w + PPT_{fixed}\right] H_{old} \quad (10)$$

A final message, Message (11), is sent from the old BS to the crossover point to complete the teardown of the old leg of the connection. The delay incurred by this message is computed identically to that of Message (10).

$$T_{11} = T_{10} = \left[\left(\frac{S_{ctrl}}{BW_w}\right) + L_w + PPT_{fixed}\right] H_{old} \quad (11)$$

Using the time spent transmitting, forwarding, and processing each of the control messages. we can compute the values of the various metrics. $T_{disrupt}$, the amount of time during which network service on the downlink to the MH is disrupted, is the time the MH needs to acquire a wireless channel, have the new BS arrange for data forwarding, and for the MH to receive the first forwarded data packet.

$$T_{disrupt} = T_1 + \left(\frac{S_{ctrl}}{BW_{wl}}\right) + T_3 + T_4 + \left(\frac{S_{data}}{BW_{wl}}\right) + L_{wl} \quad (12)$$

The amount of buffering required by the MH is determined by the amount of time during which the MH cannot transmit data on its wireless uplink. This includes the time for the MH to greet the new BS and the time for the new BS to acknowledge the greeting.

$$B_{mh} = (T_1 + T_2)\, BW_{wl} \quad (13)$$

The amount of buffering required on the old BS for downlink buffering is determined by the amount of data that will need to be "replayed" to the MH through the forwarding channel and the new BS. This data is precisely that which would have been received while the new BS was arranging for data forwarding from the old BS and waiting for the corresponding acknowledgment, plus the data that was being transmitted along the wireless link from the old BS to the MH when the MH moved into the new cell. We assume that the first forwarded data packet immediately follows the acknowledgment of the forwarding request (Message 4).

$$B_{bs,down} = \left(T_1 + \left(\frac{S_{ctrl}}{BW_{wl}}\right) + T_3 + \left(\frac{S_{ctrl}}{BW_w}\right) + L_{wl}\right) BW_{wl} \quad (14)$$

The amount of uplink buffering needed on the new BS is influenced by two terms. The first is the time to establish forwarding (minus the time taken to send the acknowledgment for the MH's greeting and the time needed for the first data packet to be arrive at the new BS). The other term is the difference in the delay between the paths going through the old and new base stations. In the case where a network offers deterministic (or perhaps even statistical) delay bounds [Ferrari92], it would be possible to use those values in this derivation.

$$B_{bs,up} = max\left(T_3 + T_4 - (2L_{wl}), \left[L_w + \left(\frac{S_{data}}{BW_w}\right)\right] max\left(H_{ctrl} + H_{old} - H_{new}, 0\right)\right) BW_{wl} \quad (15)$$

The amount of excess bandwidth-space-time product is given by the amount of network resources allocated but not in use. This includes the bandwidth used by the channel between the new BS and the crossover point during establishment of the new partial connection and subsequent acknowledgments and the bandwidth used by the channel between the old BS and the crossover point after the switchover and before that connection has been completely deleted.

$$P_{excess} = \begin{array}{l} \left(\left\{\frac{H_{new}^2 + H_{new}}{2}\right\}\left(\frac{S_{ctrl}}{BW_w} + L_w + PPT_{fixed} + PPT_{adm}\right) + \right. \\ (T_7 + T_8 + T_9) H_{new} + \\ T_{10} H_{old} + \\ \left.\left(\frac{H_{old}^2 + H_{old}}{2}\right)\left(\frac{S_{ctrl}}{BW_w} + L_w + PPT_{fixed}\right)\right) BW_{wl} \end{array} \quad (16)$$

The bandwidth-space-time product required for forwarding data from the old BS to the new BS during handoff is dictated by the amount of data that needs to be forwarded.

$$P_{fwd} = \left[T_1 + \left(\frac{S_{ctrl}}{BW_{wl}}\right) + T_3 + \left(\frac{S_{ctrl}}{BW_w}\right) + T_5 + T_6 + T_7 + T_8 + T_9 + \left(\frac{S_{data}}{BW_w} + L_w\right) H_{old} + \left(\frac{S_{data}}{BW_{wl}} + L_{wl}\right)\right] BW_{wl} H_{ctrl} \quad (17)$$

4.3 Comparison of Schemes

We have derived formulae for the performance metrics for the FR, IR, and MB algorithms, with and without the availability of hints regarding advance warning of a handoff. Substituting the values listed in Table 1 into these formulae provides the framework for our analysis. We have varied values from Table 2 to obtain values for the performance metrics listed in Table 3. H_{server} was fixed at six hops, while both H_{new} and H_{old} were varied from one to six hops, to simulate the choice of a crossover point anywhere along the path from the base stations to the server. Depending on the size and organization of the switches, the diameter implied by such a network may be as large as a small campus. For simplicity, H_{new} and H_{old} were chosen to be equal for each experiment. H_{ctrl} was chosen to be twice the distance between the crossover point and the old or new BS. For the cases where the algorithms use cell overlap hints, we assume that the MH sends the hint to the old BS one second before it enters the new cell. This value is consistent with the relatively slow movement of users.

4.3.1 Performance Results for Incremental Re-Establishment

We present a discussion of the performance results of the FR, IR and MB algorithms, both with and without hints. Graphical results are presented, however, only for the IR scheme. Our analysis of the algorithms using hints assumes the best case, where all connections have been established by the time the MH enters the new cell.

Figure 4 shows the service disruption time in the delivery of downlink data for the IR scheme. Because this metric is dependent on the time to set up data forwarding, it is highly dependent on H_{ctrl}, the number of hops in the path between the old and new BSs. The service disruption time is the same for both the case with and without hints since both request forwarding of downlink data. More generally, because forwarding of downlink data is requested in both FR and MB without hints as well as in FR with hints, the service disruption time for those algorithms is identical to that shown in Figure 4. MB with hints, which does not request downlink forwarding, yields a constant service disruption time, which is longer than the disruption shown in the figure for $H_{old} = H_{new} = H_{ctrl}/2 = 1$, but shorter than the disruption time for larger hop counts.

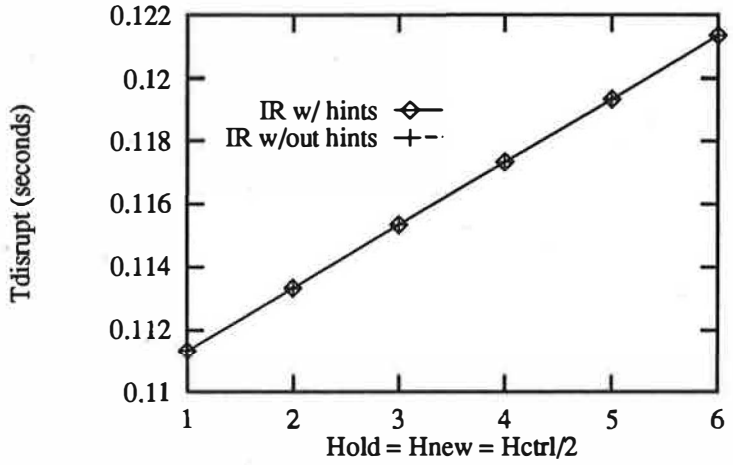


FIGURE 4. Service Disruption Times for Incremental Re-Establishment

Figure 5 displays the mobile host buffering required to temporarily store uplink data from the MH during the time which it is out of communication with any BS. Because this time includes only the greeting message and acknowledgment, the buffering required at the MH is constant over all crossover point topologies with or without the availability of hints. Because of its definition, the MH buffering is constant over all of the handoff algorithms for all combinations of topology and hint availability.

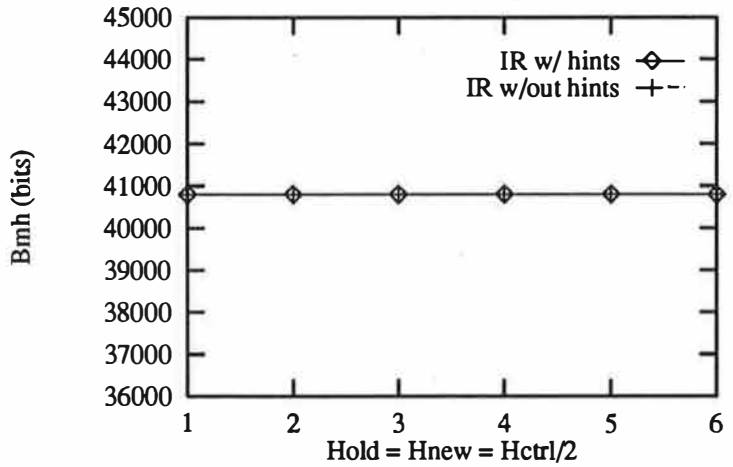


FIGURE 5. Mobile Host Buffering Requirements for Incremental Re-Establishment

Figure 6 illustrates the buffering required on the base station for downlink data. As in the case for service disruption, this metric is highly dependent on the time required to initiate data forwarding from the old BS to the new BS. Because each scheme (with the exception of MB with hints) requests this forwarding, the buffering requirements shown in Figure 6 apply to all of these schemes. The constant downlink BS buffering required for MB with hints (to buffer data sent over the radio communication switchover period) is several thousand bits less than that required for the case of $H_{old} = H_{new} = H_{ctrl}/2 = 1$ shown in the figure.

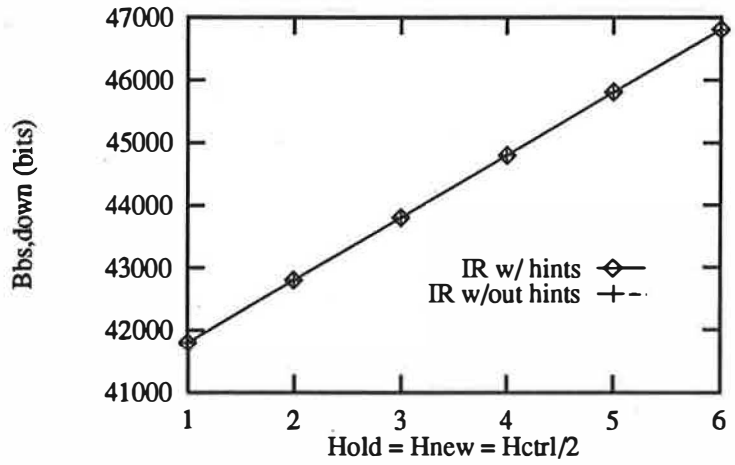


FIGURE 6. Base Station Buffering Requirements (Downlink) for Incremental Re-Establishment

The buffering required on the base station for uplink data is shown in Figure 7. For the case when hints are unavailable, this quantity is given by the maximum of the time to set up forwarding and the difference in propagation times to the crossover point along the uplink forwarding path and the new uplink path. Thus, the uplink buffering requirements are directly proportional to the number of hops along the forwarding path between the old and new BSs. The buffering requirements for FR and MB without the availability of hints are identical to those shown without hints in Figure 7.

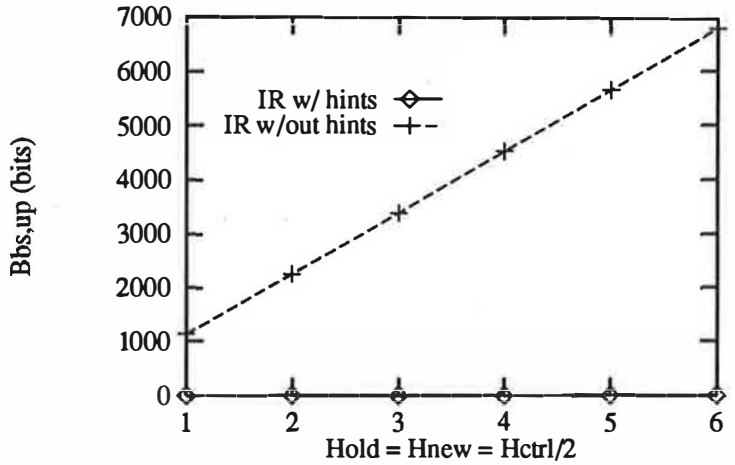


FIGURE 7. Base Station Buffering Requirements (Uplink) for Incremental Re-Establishment

With the availability of hints, none of the algorithms forward uplink data; as a result, the uplink buffering requirements with hints are dependent only on the difference in propagation times to the crossover point along the old and new uplink paths, including the time for the MH to switch cells. Because of the constant relationship between the hop counts used here, this difference is non-positive. Hence, there is no buffering required for uplink data in the hinted IR, FR, and MB cases.

Figure 8 diagrams the amount of bandwidth over all links consumed by channels established, but not in use, during a handoff. This value also reflects the duration for which these resources are held. As expected, the excess resources consumed for the case where hints are available exceed those used for the non-hinted case, because the new channel is established in advance through the use of the hint. This effect is magnified as the distance from the crossover point increases because the amount of excess resources held grows as the square of both H_{old} and H_{new}. This same behavior is exhibited by the MB algorithm. Because the resources reserved for the FR scheme are dependent only on the distance between the BS and the server, the plot of P_{excess} for FR is constant for both the non-hinted and hinted cases.

The use of hints in the IR algorithm results in a nearly ten-fold increase in the resources reserved, but not used, in satisfying a handoff. This factor is highly dependent on the choice of the time value representing how far in advance the hint is received before the MH actually moves into the new cell.

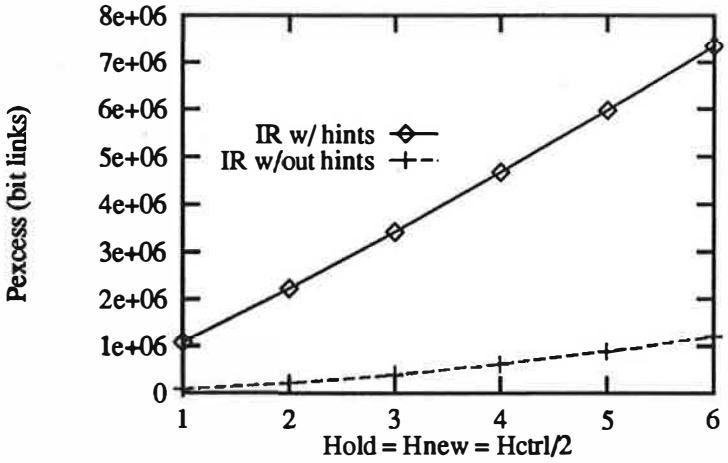


FIGURE 8. Excess Bandwidth-Space-Time Utilization for Incremental Re-Establishment

The amount of bandwidth-link-time resources consumed to perform forwarding is shown in Figure 9. The plots for the cases both with and without hints grow as the distance from the crossover point increases. This behavior is due to the fact that as H_{ctrl} increases, the number of links used to perform forwarding grows. In addition, the amount of data to be forwarded (or time that the resources will be consumed) grows as the time to set up the forwarding increases. The resources used in forwarding data for the case with no hints exceed those used in the case with hints because data must be forwarded during connection establishment for the former scheme. The use of hints results in a decrease in the resources consumed by a factor of two (for small values of the network topology parameters) to three (for larger hop count parameters). This behavior is also exhibited for the FR and MB schemes without hints and the FR scheme with hints. No data is forwarded for the hinted MB scheme.

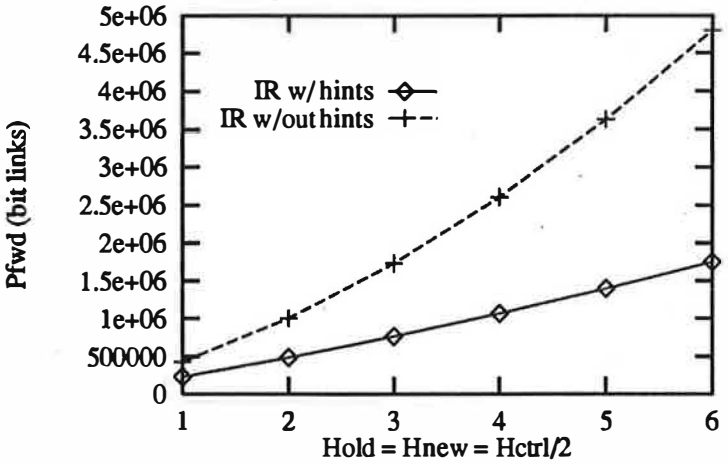


FIGURE 9. Forwarding Bandwidth-Space-Time Utilization for Incremental Re-Establishment

4.3.2 Analysis of Bandwidth-Space-Time Utilization

We next compare the bandwidth-space-time metrics of P_{excess} and P_{fwd} for the three handoff algorithms. As noted before, these metrics measure the amount of bandwidth used over all of the links and the duration of that bandwidth's use. Figure 10 shows the excess bandwidth consumed during a handoff which proceeds

without the availability of hints. Because the resources reserved for the FR scheme are dependent only on the distance between the BS and the server, it does not vary significantly with the length of the control channel. The IR and MB algorithms, however, require increasing amounts of excess bandwidth as the paths from the base stations to the crossover point gets longer. As shown in the figure, a considerable (e.g., five-fold or more) reduction in the resources consumed is possible with the use of the IR or MB algorithms instead of FR for topologies with crossover points close (i.e., within one or two hops) to the BSs. This reduction also depends on the pre-existing control path between BSs being relatively short.

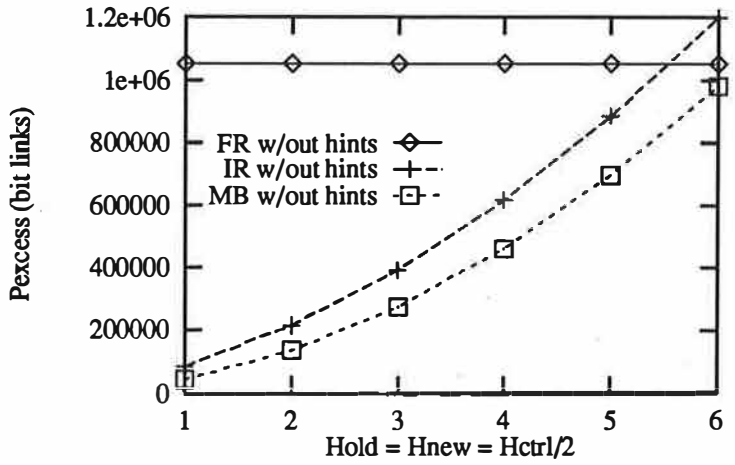


FIGURE 10. Excess Bandwidth-Space-Time Utilization for All Schemes (Without Hints)

Figure 11 shows the corresponding values of P_{excess} in the case that hints of upcoming handoffs are available. The bandwidth space-time product is linear for FR, as the time to set up the new connection from the new base station to the server is constant for all cases examined. For IR and MB, however, the length of the channel establishment varies, resulting in the nearly linear increase in P_{excess}. Again, the use of IR or MB over FR results in a considerable reduction (e.g., by a factor of three to seven) in resource consumption when the crossover point is close to the BSs and the BS-to-BS control path is relatively short.

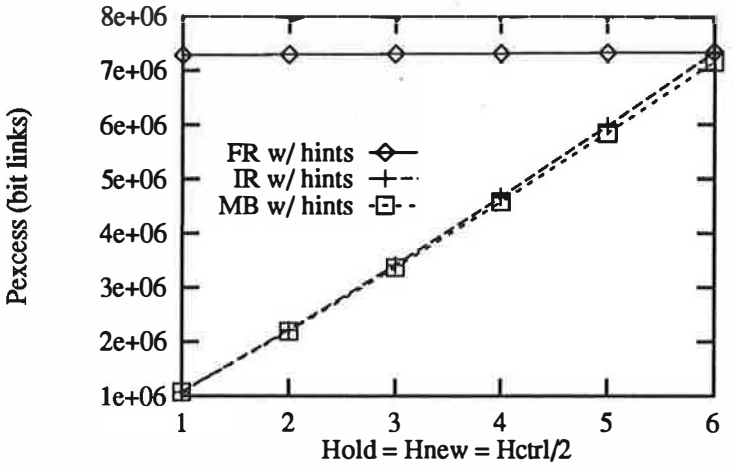


FIGURE 11. Excess Bandwidth-Space-Time Utilization for All Schemes (With Hints)

In Figure 12, the values of P_{fwd} is shown for all algorithms, assuming no advance warning hints are available. As the length of the required channel establishment is constant, the forwarding bandwidth needed to support the FR algorithm is roughly linear with the length of the control channel. However, the length of the control channel is proportional to the length of the channel establishment required for the IR and MB algorithms. The result is that P_{fwd} for the IR and MB schemes increases roughly with the square of the length of the control connection. As for previous metrics, the use of IR or MB over FR for a network with logically

adjacent BSs and crossover points decreases the amount of resources consumed in forwarding data. This decrease is 30 to 50 per cent for cases when $H_{old} = H_{new} = H_{ctrl}/2$ is one or two.

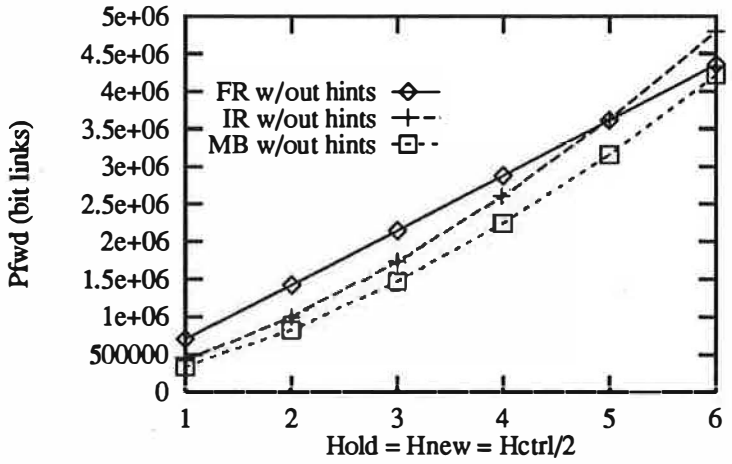


FIGURE 12. Forwarding Bandwidth-Space-Time Utilization for All Schemes (Without Hints)

Values of P_{fwd} for all three algorithms using hints are shown in Figure 13. We note that P_{fwd} is zero for the hinted scenario applied to the MB algorithm, as no forwarding of data is done in this case. The forwarding resource consumption for FR and IR differ only in the amount of work that must be done to delete the old channel. In the cases where the crossover point is close to the BSs, IR offers a 12 to 15 per cent decrease in resource consumption over FR.

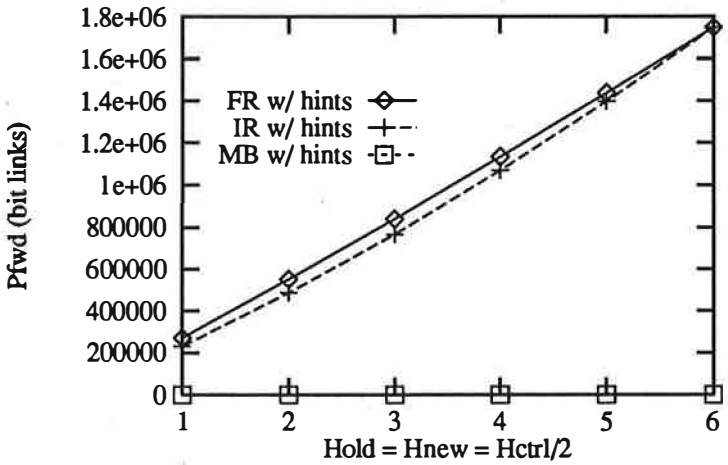


FIGURE 13. Forwarding Bandwidth-Space-Time Utilization for All Schemes (With Hints)

4.3.3 Summary of Results

When compared on the basis of service disruption time, MH buffering, or BS buffering, the FR, IR, and MB algorithms are not significantly different. One exception is the case where MB uses hints to perform a handoff, as it does not request downlink data buffering. The algorithms are significantly different, however, on the basis of excess resource consumption and resource consumption for forwarding downlink data. Using these metrics, the MB algorithm generally performs better than the other two schemes studied, for both the cases in which hints were and were not available. MB outperforms FR due to the decreased size of the channels that it must manage. It is more efficient than IR in that it does not require as complicated a channel establishment scheme.

The effects of hints on the performance of the algorithms was striking. Depending on how far in advance a hint is available before the MH moves, the excess resources necessary to complete the handoff may be significantly greater than those needed in the case where hints are not available. However, the use of hints can result in a two- to three-fold decrease in the resources necessary to forward data. In addition, the use of hints may significantly decrease the amount of BS buffering required to support in-order delivery of uplink data.

These results also show that the effects of network topology are important. If the network is constructed such that the paths between the BSs and the crossover points are short, significant reductions in the resources required for handoffs may be realized for the IR and MB algorithms over the more naive FR algorithm. This effect is also dependent on the distance between physically adjacent BSs, as control data and forwarded data must travel along this path. These results suggest that it is advantageous to place physically adjacent base stations logically close together in the wired network topology used to support mobile hosts.

5.0 Conclusions

Two recent trends in computer systems are multimedia applications and mobile computing devices. A combined multimedia, mobile computing environment poses new problems in networks. Multimedia applications typically require certain qualities of service from network services; real-time network services require connections in order to provide real-time performance guarantees. In order to provide such services in a network with mobile hosts, the network must reroute connections as hosts move between cells.

We present several schemes for supporting connection-based services in mobile networks. As a basis for comparison, we use a Full Re-Establishment algorithm, which establishes entirely new connections every time a host moves. In contrast, we present the Incremental Re-Establishment algorithm, which modifies an existing connection by establishing only the portion of the channel between the base station and the node where the old and new channels would diverge. Our Multicast-Based scheme relies on network-layer multicasting to deliver data to more than one base station during a handoff.

We have performed an evaluation of these handoff schemes using a simple analytically derived model. Our analysis of these algorithms implies that new schemes for re-establishing connections in mobile networks can provide improved performance compared to naive algorithms. In particular, the use of multicast services to support mobile host handoff may yield considerable benefits. The use of cell overlap information to facilitate handoffs may greatly reduce the amount of buffering required, at the cost of an increase in the excess resources consumed in satisfying the handoff. Our results also show that network topology is an important consideration in the design of mobile, connection-based networks. In particular, it is advantageous if physically adjacent base stations can be located with logical locality in the network.

6.0 Future Work

Since the human eye can easily interpolate missing video information, we will explore the possibility of eliminating the forwarding of downlink data during a handoff. We anticipate that this will have a significant impact on the service disruption time, resource consumption, and potentially BS buffering. In addition, we will examine our algorithms' behavior in cases where the full benefit of hints cannot be used (for example, if a cell transition follows its corresponding hint so closely that the network does not have time to complete all the processing of the hint). We are currently completing the construction of a trace-driven simulator using the Ptolemy simulation environment [Buck92], which will be used to evaluate the algorithms' performance. In addition, the simulation will quantify the system's capacity for supporting mobile hosts and processing handoffs for a variety of user data traffic and user movement patterns.

We also plan to use protocol analysis software such as Spin [Holzmann91] to formally verify the correctness of the FR, IR, and MB protocols. Making physically adjacent base stations logically close means that any changes in connection state (such as rerouting) will take place close to the handoff site, and that the effect on the rest of the network should be minimized. Our longer term research plans include studies of network topology and real-time guarantees. Much of the work done on providing real-time guarantees using resource

reservation needs to be modified to support rerouting as in [Parris92]. We hope to investigate how to incorporate real-time guarantees into our handoff algorithms.

7.0 Acknowledgments

The authors gratefully thank Ron Widyono of UC Berkeley and Doug Terry of Xerox PARC for their helpful comments and suggestions.

8.0 References

[Brodersen93] R. Brodersen, personal communication, Berkeley, CA, June 1993.

[Buck92] J. Buck, et al. "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal on Computer Simulation*, special issue on "Simulation Software Development", 1992.

[Ferrari90] D. Ferrari. "Client Requirements for Real-Time Communication Services", *IEEE Communications Magazine*, Vol. 28, No. 11, November 1990, pp. 65-72.

[Ferrari92] D. Ferrari, A. Banerjea, and H. Zhang. "Network Support for Multimedia—A Discussion of the Tenet Approach", Tech. Rept. TR-92-072, International Computer Science Institute, Berkeley, CA, November 1992.

[Holzmann91] G. Holzmann. *Design and Validation of Computer Protocols*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Ioannidis91] J. Ioannidis and G. Maquire, Jr. "The Design and Implementation of a Mobile Internetworking Architecture", *Proc. 1993 Winter USENIX*, pp. 491-502.

[Mah93] B. Mah. "A Mechanism for the Administration of Real-Time Channels", MS Report, Tech. Rept. UCB/CSD-93-735, University of California, Berkeley, CA, March 1993.

[Parris92] C. Parris, H. Zhang, and D. Ferrari, "A Mechanism for Dynamic Re-routing of Real-time Channels", Tech. Rept. TR-92-053, International Computer Science Institute, Berkeley, CA, August 1992.

[Schilling91] D. Schilling, et al. "Broadband CDMA for Personal Communications Systems", *IEEE Communications Magazine*, Vol. 29, No. 11, November 1991, pp. 86-93.

[Schroeder92] M. Schroeder, personal communication, Berkeley, CA, October 1992.

[Sheng92] S. Sheng, A. Chandrakasan, and R. Brodersen. "A Portable Multimedia Terminal", *IEEE Communications Magazine*, Vol. 30, No. 12, December 1992, pp. 64-76.

[Teraoka91] F. Teraoka, Y. Yokote, and M. Tokoro. "A Network Architecture Providing Host Migration Transparency," *Proc. SIGCOMM '91*, pp. 209-220.

[Zhang92] H. Zhang and T. Fisher. "Preliminary Measurement of the RMTP/RTIP", *Proc. Third International Workshop on Network and Operating System Support for Digital Audio and Video*, San Diego, CA, November 1992.

Agent-Mediated Message Passing for Constrained Environments

Andrew Athan and Dan Duchamp
Computer Science Department
Columbia University
New York, NY 10027
{athan,djd}@cs.columbia.edu

1 Introduction

We outline desirable features for a general-purpose inter-process communication mechanism suited to the needs of mobile computers operating under bandwidth and power constraints.

1.1 World View

Since mobile computing is presently still in its infancy, we first discuss our "world view" and how it leads to the problems that our design seeks to solve. A convenient oversimplification of the design space for mobile computing is the following:

- There is a spectrum of mobile devices, anchored at the two ends by the notions of "terminal" and "computer."

  A "terminal" is a device with limited ability for computation or storage. A terminal provides a user interface only, and is animated entirely by signals from some "infrastructure" computer. A computer, on the hand, possesses some degree of compute and storage capacity, and is capable of standalone operation.

- There are two sorts of communication networks, public and private. The essential difference is that public networks charge per-use and therefore communication is a recurring cost, whereas in private networks the cost is largely front-loaded to a one-time installation, with limited ongoing cost paid for network maintenance, and typically none for communication.

  Although the public/private distinction need not imply anything about available bandwidth, in practice public networks — both wired and wireless — are likely to offer very much less bandwidth per user than private networks during the foreseeable future. Furthermore, in many cases the "last link" to the client computers can be expected to be wireless. In these cases, bandwidth will be limited whether the network is public or private. The resulting architecture is a highly capable infrastructure of high speed networks and servers, fringed by a much lower speed and more expensive last link to the mobile computer.

Because of their complete dependence on access to bandwidth for communication with the infrastructure, terminal-based system designs seem ill suited for operation in low-bandwidth public-network environments. Accordingly, because of their wider applicability, we focus our attention on computer-based designs.

2 Design

Given that mobile devices will commonly operate as computers, there is need for application and system software running on a mobile computer permitting the exchange of data between mobile

computers and servers in the infrastructure. Constraints on the mobile computer's communication bandwidth and power consumption affect how best to perform this communication. We contend that current binding and message delivery mechanisms are not ideal for these circumstances.

2.1 Failings of Current Mechanisms

2.1.1 Overuse of Bandwidth

If the "last link" to the mobile computer is a wireless one, then it is desirable to limit the use of that link's bandwidth. One approach is to recognize that not all data returned out of the infrastructure is equally valuable. Present transport mechanisms view intra-message data as a one-dimensional stream, and flow control governs only which prefix of the stream is sent. Likewise, separate messages (inter-message data) are viewed as equally valuable, and cannot be filtered except once received by the destination. The architecture discussed above — a high speed infrastructure fringed by a lower speed and more expensive last link — motivates the addition of a policy module at the edge of the infrastructure that can reorder, suppress, or otherwise process data flowing out of the infrastructure to mobile computers.

2.1.2 Client-Server Model

With current distributed binding techniques, a server registers a *(string, port)* tuple with a name server; the client later looks up the tuple and remembers the port value, thus binding to the server. Except for some replicated services or administratively pre-arranged "well known ports," the lifetime of a binding is typically equal to the lifetime of the particular server process that registered the tuple.

Another aspect of this model is that a server must be running before a client can get service. As a result, server daemons tend to proliferate, one for each service. In a workstation/office environment, such proliferation is not unreasonable: the workstation is probably always turned on and connected to the network; furthermore, the workstation probably has virtual memory and plenty of backing store, meaning that accretion of a large number of daemons that seldom do anything has little cost. When communicating with a mobile computer, the situation may be very different. It may be infeasible — for power reasons — to expect the mobile computer to be continuously running all the various daemons that might be addressed by incoming messages: the mobile computer may be turned off, or it may not have virtual memory or may be limited in backing store; either way, accretion of many long-lived server daemons may be impossible or undesirable.

It is preferable to automatically start applications on demand. Simple examples of this capability exist: the UNIX *inetd* program, for example. It should also be possible to pause and automatically resume applications as well as start them. During the time that an application at one endpoint of a transport connection is paused, the connection should not break if the other endpoint sends data.

2.2 Overview

We propose an inter-process communication mechanism that addresses the shortcomings noted above. Specifically,

1. A message is considered not as a *sequence* of typed objects (as in Mach [1], for example), but as a *hierarchy* of typed objects. Roughly speaking, "important" or immediate-delivery data belongs near the top of the hierarchy, while less important or deferrable data belongs near the bottom.

2. An undeliverable message can be stored for later delivery. Storage includes the possibility of filtering and/or reordering message delivery, to place important data ahead of less important data.

3. Delivery of a message destined to a server program that is not running can cause the destination program to be automatically started or resumed.

Each of these features is governed by "hooks" made available to both source and destination. These are *policy* hooks that help customize message delivery. Policy issues include how to perform message filtering and whether/when to automatically start a program on the mobile computer. This is the much-discussed idea of an "agent" that continuously operates within the infrastructure on behalf of a mobile user or computer, and building such an agent is the goal of our work.

The ability to start the destination process on demand and/or to store a message for later delivery makes our model of binding more general than the standard client-server model. In the client-server model, both source and destination must be executing simultaneously. The binding becomes useless when either process shuts down. In contrast, our model includes Email-like *deferred communication*, which is intended to help cases wherein power or bandwidth limitations might render communication with a mobile computer sometimes impossible and other times undesirable.

The following sections sketch our ideas in somewhat greater depth.

2.3 Hierarchical Data

Organizing data into a hierarchy is a commonly proposed technique for the coding and framing of images and video for network transmission. In this formulation of the idea, data at the top of the hierarchy comprises a moderate-resolution basis of the image(s) while data at the bottom of the hierarchy represents additional information that adds possibly-nonessential sharpness to the image(s). By transmitting more or less of the hierarchy, image quality can be varied smoothly according to available bandwidth. Other domains in which it can be useful to organize message data into a hierarchy include:

- Email, which can be divided into portions such as: important header fields, unimportant header fields, message summary, new versus quoted text, checksum, and annotations and/or large embedded objects (bitmaps, sound) that might be considered optional.
- Ordinary documents, which often divide naturally into a hierarchy. Hypertext documents are inherently composed of a collection of clearly separate units.
- File data, especially that requested by an interactive editor. In this case, it is important to deliver the first page promptly, but remaining pages can often be delayed.

In each case, the goal of the hierarchy is to distinguish vital data from incidental data.

In our formulation, the "importance hierarchy" is specified by the source and made visible to the destination, so the destination can choose which parts of a message to ignore, which parts to receive, and when to receive.

Piece-by-piece receipt of messages already organized into a hierarchy is a conceptually simple extension to primitives for typed messages. For example, Mach's message format is a fixed-size header followed by a variable number of descriptor/data pairs; each (fixed-size) descriptor defines the size and type of the data (either in-line or out-of-line) that follows. Mach's `msg_receive()` primitive receives an entire message into a pre-allocated buffer. A different message format similar to an expanded i-node is appropriate for hierarchical data. The new message format contains a fixed-size header, a small fixed-size area for descriptor/data pairs, and a fixed number of conceptual "pointers" to the remainder of the message. Each such pointed-to segment contains further descriptor/data pairs. The new `msg_receive_segment()` primitive receives one segment as a time; the existing `msg_receive()` can still be used to receive the entire message. The descriptor format is expanded to include an additional field that can describe the meaning of the data. This information can help guide the receiver as to which segments to receive next; Mach's descriptor currently carries only type information.

One of the trickiest issues in formulating hierarchical messages is defining the moment the kernel may consider a message as delivered. This atomic transition point is important for permitting the operating system to begin garbage collection and other actions that are dependent on the message having been delivered.

2.4 Binding and Message Delivery

The address of a communication endpoint on a mobile computer consists of two sub-addresses, one being the intended destination, and the other being the agent that lives in the infrastructure as a permanent representation of a mobile user or host. This two-part address we call a *composite address*.

The intended destination sub-address is statically assigned so that it can be long-lived. We presume that a program running on a mobile computer might be frequently restarted and hence would receive different port numbers on each instantiation. Therefore, a port is a poor address for the intended destination because frequent re-registration and re-binding would be necessary. The agent's sub-address is an ordinary Mach port.

The endpoint address consists of *both* the long-lived sub-address and the agent's port since one always needs the combination of the two in order to successfully reach either the destination or, as a backup, its agent. This address format obscures the port of the destination on the mobile computer, since it is regarded as information possibly too short-lived to be of practical use. The composite address is presumed to be far more long-lived, because the agent presumably restarts less often — its very purpose is to be available as a proxy for the mobile computer.

To learn a composite address, a process either has the address passed to it (e.g., as the return address included as part of a request) or goes to a new type of name server that associates a string or other identifying pattern with the composite address.

Messages sent from the mobile computer are addressed to an ordinary Mach port, and list the composite address as the return address. A message sent to the mobile computer is addressed to the agent's port sub-address embedded within the composite address; accordingly, messages traveling from the infrastructure to the mobile computer are passed through the agent, where they can be delayed, suppressed, etc.

Once such an "outbound" message passes the agent's policy tests and merits delivery, the agent must know an address on the mobile computer. The intended destination may be up and running, in which case the agent must discover the Mach port value for the destination; alternatively, the agent may need to start the appropriate program on the mobile computer. Accordingly, the mobile computer must run a Mach name server (which, by definition, has a well-known port value) that is augmented with an interface to the agent and the ability to fork servers on demand. This name server stores the string-to-port mappings for the active programs on the mobile computer; the binding is short-lived, as noted in the criticism above, but this short-lived binding is visible to the agent only, and therefore is a limited inconvenience. Additionally, the agent maintains a database listing the names and file locations of server executables.[1] The agent uses this information to direct the mobile computer's augmented name server to start a program on demand.

In summary, besides the need for the agent itself, the mobile computer must run an augmented names server, and a new name service must be created to maintain associations between strings and composite addresses.

3 Related Work

The ideas described in this paper are only a small step away from many different pieces of prior work.

The notion of automatically starting the program that is a message's destination is present in

[1] Much like **nanny**, Mach's server-server.

various "software toolbus" designs in the software engineering community. One toolbus with this capability is Polylith [4]. Tadpole portable computers have for some time run a modified version of SunOS that can pause and resume the whole system. While this mechanism is too coarse-grained for our needs, pausing the whole system is related to pausing a process, since process state is not self-contained – important state exists in kernel data structures.

Structuring message data is an old idea. Many message-based operating systems provide a structured, typed message model; Mach is only one example. Typed, structured data is the basis for several standards, including the ASN.1 message syntax and ODA. In these cases, however, typing is added to structure in order to ease (or make possible) data interpretation by the application.

The notion of separating logical "delivery" from physical, on-demand delivery of data is the basis for copy-on-write and, at finer granularity, on-demand RPC parameter passing [3].

There are numerous naming/binding mechanisms for message-based systems. Two differing examples include Mach, in which an endpoint is named by a string and bound to a port, and Field [5], in which an endpoint is named by a printf-like pattern and binding is done by a central agent that matches messages sent with patterns registered earlier by interested destinations.

Mach contains the notion of a "backup port" [2]. The right to receive messages reverts to a "primary" port's backup port when the primary port is destroyed. This mechanism is close to something we need, since it provides some capability for an agent to receive messages intended for a now-unreachable mobile computer. However, our approach requires a more long-lasting relationship between an endpoint on a mobile computer and the corresponding endpoint at the agent. In our setting, the current locus of delivery can ping-pong back and forth between the mobile computer and the agent.

4 Summary

We have presented a case for the need for an inter-process communication mechanism that provides (1) a backup message delivery location where message storage and filtering can be done, (2) the ability to receive data partially and on-demand, and (3) the ability to sense lack of a server and initiate its execution if warranted. We have sketched a simple design (for a Mach environment) that can implement these features.

All these capabilities are motivated by the likelihood that mobile computers will be operating in bandwidth-limited and power-limited environments. These constraints raise the cost of extraneous communication between the mobile computer and the infrastructure, and motivate mechanisms to distinguish between essential and non-essential communication. The likelihood that a user will opt to disconnect his computer from expensive public communication facilities motivates deferred communication, which converts what would otherwise be a binding error into delayed, elective message delivery.

5 References

[1] M. Accetta et al. Mach: A New Kernel Foundation for UNIX Development. In *Proc. Summer USENIX*, USENIX, pages 93–112, July, 1986.

[2] R. V. Baron et al. MACH Kernel Interface Manual. Unpublished. Available by anonymous ftp from mach.cs.cmu.edu:/mach3/doc/unpublished/sup/sup.doc

[3] Y. J. Cycowicz, J. Micallef, and G. E. Kaiser. Demand-Driven Parameter-Pasing in Remote Procedure Call. Unpublished technical report. February 1987.

[4] J. M. Purtilo and C. R. Hofmeister. Dynamic Reconfiguration of Distributed Programs. In *Eleventh Intl. Conf. on Distributed Computing Systems*, IEEE, pages 560–571, May 1991.

[5] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(4):57-66, July 1990.

Local Area Mobile Computing on Stock Hardware and Mostly Stock Software

Terri Watson and Brian N. Bershad

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Abstract

In the Fall of 1992, the graduate Operating Systems class at Carnegie Mellon University implemented the necessary components to provide applications with a programmable interface to a mobile palmtop computer. The goal of the project was to expose project members to the area of mobile computing through "shock immersion." Over the course of two months, students designed and implemented the infrastructure for a simple mobile computing environment for low-end palmtop machines. This programming environment was used to develop a suite of mobile applications such as a mailer, graphical locator map, the game tetris, and a scheme interpreter that allowed functions to be remotely executed on the palmtop. In this paper, we describe the results of the course project and the lessons learned.

1 Introduction

This paper describes a local area mobile computing system that was built by the students in a graduate operating systems course. The goal of the project was to address many of the issues that arise when dealing with mobility of low-end palmtop computers (really, simple user interface devices) in the context of a single insulated environment (a building). In meeting this goal within the time frame of a single semester course, we used available hardware and software, and well-understood systems building techniques. This approach allowed us to concentrate on the issues of communication, routing, and interface design, while leveraging off of equipment already on our desks, or available inexpensively at a local store.

The BNU system is the result of the course project. BNU builds on an existing Unix network infrastructure to support small scale mobile computing for inexpensive, physically compact devices, such as palmtops, or "wimps" (wireless interconnected mobile processors). BNU supports wimp mobility through the use of proxy processes that run on workstations in the local area network. Each wimp has a proxy representative that presents a fixed location to applications. An RPC system provides reliable, sequenced communication between wimps, which are carried, applications, which are run on workstations, and proxies, which bridge the gap between the mobile and non-mobile world. Messages are automatically forwarded through dynamic adjustment of route information as wimps relocate. A nameserver acts as an initial contact point for location information during application bootstrapping.

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035, by the Xerox Corporation, and by Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award. Watson was partially supported by a National Science Foundation Graduate Fellowship.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, XEROX, DEC, the NSF, or the U.S. government.

In this paper we describe the BNU system, and a set of mobile applications that demonstrate both the power and limitations of our prototype. The rest of this paper is organized as follows. In Section 2 we cover the mobile computing model supported and system structure. In Section 3 we describe sample applications. In Section 4 we summarize some lessons learned from the project. In Section 5 we discuss related work. Finally, in Section 6 we present some conclusions.

2 BNU structure

Figure 1 illustrates the main components of the BNU system. *Wimps* are mobile user interface devices with capabilities for limited computation. *Proxies* serve as the wimps' presence in the LAN, concealing from both wimps and applications many of the complicating aspects of mobility. All application communication with a wimp is handled though that wimp's proxy. *Routers* are processes running on the machines to which wimps become directly connected. They ferry the packets from the mobile network (in our case, a low-bandwidth infrared link, or a tethered serial line), into the LAN. *Applications* are the end service providers. They run on workstations and communicate with wimps through proxies.

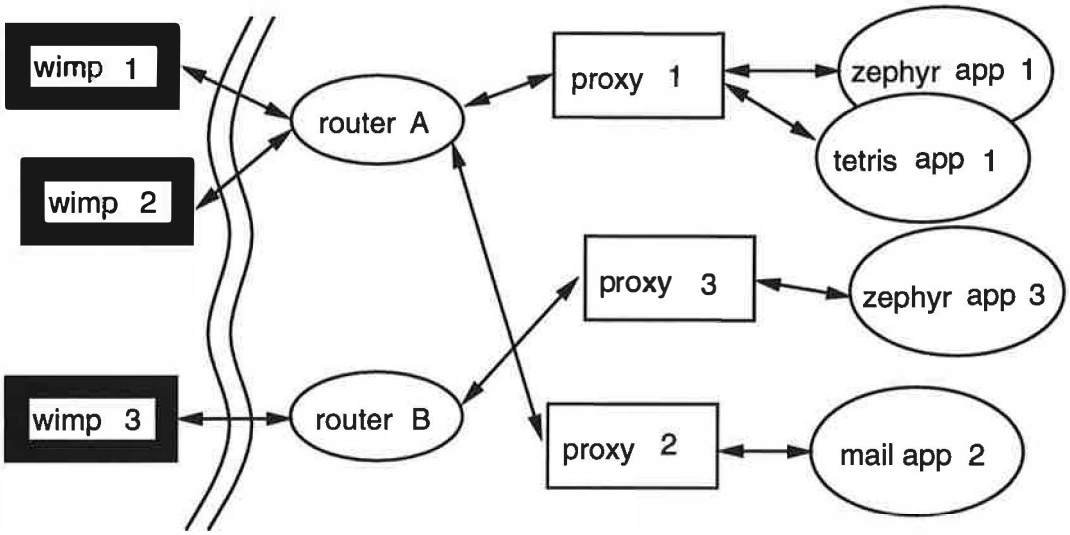


Figure 1: *BNU components. Wimps connect into the LAN by way of router processes that forward messages to the wimp's proxy. Proxies are the intermediaries between wimps and applications, hiding the complications of wimp mobility from both.*

BNU applications execute on workstations in the LAN, which are connected via a slow link to a mobile user interface device. The wimps are capable of local processing; applications make use of this by extending the wimp's RPC interface with additional functions that can be downloaded dynamically. In the rest of this section we discuss issues relating to communication, the wimps themselves, proxies and routers.

2.1 RPC and naming

Applications, proxies, wimps, and routers communicate through BNU RPCs. An RPC interface is exported by registration with the nameserver and imported via nameserver lookup. We built a simple interface definition language and implemented an interface generator for

BNU called BIG[1]. BIG allows the use of arrays, structs, and typedefs in the interface description. Parameters are tagged as **in, out**, or **inout** and procedures as **with_reply** (synchronous) or **no_reply** (asynchronous).

Bandwidth and latency limitations of the communication link to the wimp encourage the use of function shipping. This allows applications to dynamically extend the wimp RPC interface with custom procedures. Multiple operations can be accomplished with one RPC and local processing on the wimp, requiring less bandwidth. Another motivation for providing an extensible RPC interface is the increased likelihood of short periods of disconnection, or infrequent high latency RPCs. The latter may occur when moving from one communications cell to another, as processes in the LAN (proxy, routers) become aware of the location change and update their routing information. The ability to load certain routines and data on the wimp can mask these glitches which play havoc with time-crucial applications (e.g. games) or just generally annoy the user.

To permit such extensions in BNU, we modified a scheme interpreter to allow the execution of an *eval* call in the scope of a remote scheme interpreter executing on the wimp. An interpreter on the application/workstation side includes a remote *eval* function that sends an expression to an interpreter on the wimp where the actual evaluation is performed. This permits the definition and subsequent execution of new functions on the remote side, effectively extending the exported RPC interface of the wimp.

Each RPC is directed towards a named endpoint in the BNU environment. Specifically a "BNUid" must be provided as the first argument to an RPC. BNUids are similar both to internet socket addresses, in that they identify an endpoint of communication, and to ports in a system like Mach [Draves90] in that they are location transparent and can therefore name mobile endpoints.

A name in the BNU namespace is composed of two parts: a domain and an instance. Each user has their own domain, much like a UNIX userid, and entities that exist on behalf of that user are instances within that domain. The wimp and proxy are well-known instances within a domain. Other instances are assigned dynamically to applications as they are started. The well-known instances, like well-known TCP ports, allow new applications to determine the BNUids of essential entities in the BNU environment such as the nameserver and proxy without having to access a nameserver.

A nameserver provides service for all domains, so it has both a well-known instance and domain. The nameserver maintains mappings from key values to data, most commonly BNUids to native addresses. A native address is an endpoint address outside of the BNU naming scheme that can be used by the underlying transport protocol. For example, a native address in the LAN is an internet socket address.

2.2 The wimp

Wimps are the mobile user interface devices that tie users into the BNU computing environment. Our wimps are HP 95LX palmtops running MS-DOS. They each have 1MB of RAM, a 300–19,200 baud serial line, a 2400 baud short-range infrared (IR) link, one PCMCIA 1.0 slot, and a 16x40 character display with graphics mode. The growing demand for small computers with PC-like capabilities means such devices are becoming increasingly inexpensive, on the order of a few hundred dollars.

The wimp software for our prototype system exports a simple text and graphics RPC interface to applications, capable of drawing strings, simple graphics functions (pixels, lines, cursor motion) and handling keyboard input. Much of the screen oriented code was derived from a freely available graphics interface module for the HP. The communications support provides reliable, sequenced messages and RPC handling.

[1]The name (and syntax) derives from Mach's stub generator MIG, although BIG supports only the transfer of simple types.

The wimps can be connected through to the LAN via either a 9600 baud serial link when in "tethered" mode, or a 2400 baud link when wireless. Because the HP95's builtin IR link is limited to a range of about 8 inches, we built our own unit that could transmit data up to about 8 feet. Since the IR units were built in parallel with the rest of the system, most of the time wimps ran in tethered mode.

Multiple applications may attempt to control the wimp at any one time, so BNU relies on a "focus" mechanism, similar to that used in the X Window System [Scheifler*86]. Applications request focus and are notified by RPC when the focus is acquired or lost. Only the application with focus can perform I/O operations on the wimp. The proxy records which application has current focus and simply blocks any request from those that don't.

2.3 The proxy

A proxy process represents the wimp in the network at a fixed native address. When an application is started, it contacts the nameserver to learn the route (native address) to the proxy for its domain. The nameserver will start a proxy if one does not currently exist for the calling user. The proxy in turn contacts the nameserver to determine the wimp's location. At this point, a bootstrapping application (such as a terminal and shell) is started, from which the user can invoke other applications.

Each application contacts the nameserver at its startup to locate the proxy for the user on whose behalf the application is running. Applications and their wimp direct all communication with each other through the wimp's proxy. By interfacing through the proxy, application processes are unaware of changes in wimp location and do not need updates to local routing information. The RPC interface exported by the proxy is the same interface as presented by the wimp, so an RPC to the proxy typically results in an identical RPC to the wimp.

Optimizations within the proxy enable it to hide latency from the applications and better utilize limited bandwidth. Non-blocking RPCs from applications are queued by the proxy and immediately acknowledged. This asynchrony with respect to the application allows delivery of RPCs to the wimp to proceed in parallel with application processing, potentially reducing idle link time caused by waiting for the application to generate the next RPC. Where interface semantics allow, the queueing permits multiple RPCs to the proxy to be collapsed to a single, equivalent RPC to the wimp (for example, cursor motion followed by a text write).

2.4 Routing

Physical mobility causes the location of the wimp's connection into the LAN to change, requiring that messages be re-routed. The RPC system is responsible for converting BNUids (used at the application level) to native addresses that can be used for transport and routing.

Each entity (applications, the proxy, a wimp) maintains a local hintcache that maps BNUids into native addresses. The messaging layer, when delivering a message to a particular BNUid, inspects its hintcache for a match. If no match is found, the messaging layer queries the nameserver and the result is cached. The local route cache is updated whenever a message is received, always providing the messaging layer with enough routing context to deliver a reply message.

Routers, which are processes running on machines to which wimps become electrically connected, periodically broadcast "You Are Here" packets which are received by any in-range wimps. If a wimp learns that it has moved (where it is now is different from where it was a few seconds ago), it sends a location status message to its proxy. In this way, a proxy is as aware of its wimp's location as is the wimp.

2.5 Some problems

There are some clear, obvious problems with the system structure described in this section. For example, the current system assumes that a user is only using one wimp at a time, ensuring that the wimps are no more ubiquitous than the people who use them. Further partitioning the domain component of the BNUid namespace would address this problem. Another problem is that the current nameserver is centralized and non-replicated, making it an obvious bottleneck and single point of failure. Nameservers, though, are relatively easy to replicate using any of a number of techniques and systems [Birman85]. Since our environment did not have an abundance of IR units (we built 2), the system was largely developed and used in tethered mode, so some situations, such as "what happens when somebody is bouncing between IR cells" are not adequately addressed in the implementation. Lastly, there is no real security in the system. Messages are sent in the clear, and a message asserting a fact followed by a request (such as "my name is bershad; show me my mail") is taken at face value.

3 Applications

We developed about a dozen mobile applications for BNU. Some of these applications were:

- *Zephyr.* This application is a notice transport and delivery (paging) system developed by MIT-Project Athena [DellaFera*88]. Zephyr allows the delivery of a text message to a set of users in the system. Minimal changes were required to adapt the service to the BNU environment. A simple process in the workstation environment posed as the mobile user. When it received messages through the Zephyr notification service, it forwarded them on to the wimp using BNU RPC through the proxy.

- *Graphical Locator.* This application uses location information to respond to queries for directions within a specific locale. An animated figure on the display describes a path from the current location to the destination.

- *BNUterm.* This application is a terminal emulator that, in some sense provides the most flexibility, but it is not well suited to the wimp. BNUterm, derived from a VT-100 emulator, uses a character-at-a-time interface which performs poorly over the high-latency link. Furthermore, the wimp's small screen and awkward keyboard make its usage as a terminal quite unpleasant.

- *Tetris.* This is an arcade-style game in which the player manipulates falling blocks of various shapes. This application worked well because of the extensible RPC paradigm. Tetris defined additional RPC functions to do more complex operations on the wimp (e.g. display updates) and so reduced bandwidth requirements.

BNU also includes a simulator that emulates the wimp in an X11 environment. The simulator exports the same RPC interface as do the wimps, which allowed application development to proceed in parallel with infrastructure, and facilitated early interface testing and design refinements. That all interaction with wimps was through a location transparent RPC service meant that the same application could communicate with either the simulator or the actual wimp.

4 Some lessons learned

Building the BNU system allowed us to experiment with issues such as as routing, location transparency, and the limiting characteristics of the wimp interface. The following are some of the lessons learned from BNU:

- Existing system software (e.g. MS-DOS, UNIX, threads, sockets) provides service to allow the construction of an experimental mobile palmtop system. In particular MS-DOS is small, cheap (i.e. available on inexpensive hardware), and allows use of existing software packages, such as the scheme interpreter and the display code for the HP.

- The portability of devices such as the HP palmtop creates a new application paradigm. Unfortunately, the constraints on the physical size of these devices leads to awkward user interfaces that render many common Unix applications unusable in practice. For instance, an extended editing session on the palmtop can be particularly unpleasant. While UNIX was easily adapted to the last revolution in interface technology, namely windowing, it is clear that its typical terminal-style user interface is ill-suited to the next revolution, namely small-screen mobile machines.

- Proxies work well to insulate applications from wimp location and temporary disconnection during relocation.

- RPC with transparent routing also insulates applications from the details of wimp location.

- Extensible RPC yields better application performance on low bandwidth communication links by increasing the amount of work done per RPC. For example, the Tetris game with downloaded functions ran in real time, but BNUterm, which causes an RPC per character, was painful to use.

- Function shipping to support extensible RPC increases performance, but at the cost of greater wimp state, and therefore more complicated recovery mechanisms. Currently, applications are required to support state reloading if they choose to take advantage of extensible RPC.

- Asynchronous invocation through the proxy facilitates better utilization of link bandwidth by allowing multiple application requests to be queued and delivered in parallel with application processing. This queueing allows compression of multiple RPCs and potentially reduces idle link time.

5 Related work

Our use and management of wimps in BNU parallels that of the ParcTabs in the ubiquitous computing project at Xerox PARC [Weiser91, Weiser93]. Both systems use the mobile units primarily as user interface devices, and require applications executing in the LAN to communicate with the device through a proxy (or agent) [Adams*93]. The projects differ in one main dimension: BNU was built using an off-the-shelf palmtop running MS-DOS, an industry standard operating system. This allowed us to inexpensively acquire the hardware, and to leverage off of existing software, such as I/O drivers and a scheme interpreter. In contrast, Xerox's ParcTabs are custom-built, expensive, and require a special-purpose operating system. The main advantage of the ParcTabs, however, is that they are extremely well-integrated, offering a relatively high-bandwidth IR link (19.2Kb) with low power requirements. With time, though, we expect comparable communications hardware to become inexpensively available for commodity palmtops.

The graphical locator application provides navigation functionality similar to the VuMan [Akella*92, Smailagic*93] and Navigator [Siewiorek*93] wearable computer systems. The VuMan systems allow a user to page through maps and blueprints to retrieve context-sensitive information using a "heads-up" display. The Navigator system follows on this by incorporating telecommunications, speech recognition, and position sensing.

6 Conclusions

The BNU system and applications exposed us to practical concerns in mobile palmtop computing and allowed us to experiment with some simple solutions. Our use of stock hardware and "mostly" stock software demonstrates that there exists already a large amount of distributed systems technology (and artifact) that can be leveraged against the problems introduced by this class of mobile computing.

Acknowledgments

The BNU project resulted from the combined efforts of the students in the Fall 1992 CS 712 (graduate Operating Systems) course. While the authors did most of the worrying, the following students did most of the real work: Terry Allen, Shumeet Baluja, Claudson Bornstein, Yirng-An Chen, Scott Crowder, Eugene Fink, Geoff Gordon, Tammy Green, Karen Haigh, Todd Kaufmann, Jennifer Kay, James Landay, Gerald (Rob) Malan, Will Marrero, Paul Olbrich, Robert Olszewski, Manish Pandey, Henry Rowley, Stefan Savage, Motonori Shindou, Nick Thompson, Stephen Weeks, Bob Wheeler, Hao-Chi Wong, and Masanobu Yuhara.

References

[Adams*93] Adams, N., Gold, R., Schilit, B., Tso, M. and Want, R. The ParcTab Mobile Computing System. *Submitted for publication.*

[Akella*92] Akella, J., Dutoit, A. and Siewiorek, D. P. Concurrent Engineering: A Prototyping Case Study. In *Proceedings of the 3rd IEEE International Workshop on Rapid System Prototyping*, June 1992.

[Birman85] Birman, K. P. Replication and Fault-Tolerance in the ISIS System. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pp. 79–86, December 1985.

[DellaFera*88] DellaFera, C., Eichin, M., French, R., Jedlinsky, D., Kohl, J, and Sommerfeld, W. The ZEPHYR notification service. In *Proceedings of the Winter 1988 USENIX Conference.*, pp. 213–19, February 1988.

[Draves90] Draves, Richard P. A Revised IPC Interface. In *Proceedings of the First Mach USENIX Workshop*, pp. 101–121, October 1990.

[Scheifler*86] Scheifler, R. W. and Gettys, J. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[Siewiorek*93] Siewiorek, D. P., Smailagic, A., Lee, J. and Adl-Tabatabai, A. An Interdisciplinary Concurrent Design Methodology as Applied to The Navigator Wearable Computer System. EDRC Technical Report, Carnegie Mellon University, May 1993.

[Smailagic*93] Smailagic, A. and Siewiorek, D. P. The Design and Implementation of the VuMan Wearable Computer EDRC Technical Report, Carnegie Mellon University, March 1993.

[Weiser91] Weiser, M. The Computer for the Twenty-First Century. *Scientific American*, 265(3):94–104, September 1991.

[Weiser93] Weiser, M. Some Computer Science Issues in Ubiquitous Computing. To appear, *Communications of the ACM*, July 1993.

Experiences with X in a Wireless Environment

Christopher Kent Kantarjiev
Alan Demers
Ron Frederick
Robert T. Krivacic
Mark Weiser

Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

ABSTRACT

Wireless computing is all the rage; the X Window System seems to be inescapable. We have been experimenting with the cross-product, and have had mixed results. The network may not be the computer any more, but it certainly influences the way the computer performs, and adding a fairly atypical network layer cannot help but expose some underlying assumptions. We discuss a few that we found and go on to speculate about how to best push X in the direction of mobile and location-independent computing.

Introduction

For the past few years, a number of researchers from the Computer Science Laboratory at Xerox PARC have been investigating the foundational glue for a possible next generation of computing in which each person is continually interacting with hundreds of nearby wirelessly interconnected computers. This is known as *Ubiquitous Computing* or *ubicomp* [Weiser91]. This vision contends that computational devices will disappear into the background, just as the electric motor has disappeared into the background of everyday life. We believe that these devices, potentially numbering in the hundreds per person, will be mobile, location-aware, and in communication with their environment. Thus, they can change their computation based on their current location, environment, and external events.

In order to experiment with these ideas, a series of hardware devices have been built at Xerox PARC. We segregate devices into three categories by size: inch, foot and yard. Inch sized devices range from 6 inches and smaller, including devices such as the personal display assistants that are coming to market as well as those embedded in furniture and light switches. Foot sized devices are between 6 and 18 inches. Yard sized devices can range from 18 inches to 6 feet.

It is key to the vision that applications interact and flow across the various classes of devices. One way to facilitate this is to provide a common graphical/user interface environment. Even though we don't believe it is ultimately the correct solution, we have chosen to experiment with the X Window System [Scheifler92] as this environment, both because it is a standard and because we are familiar with it. We believe (or believed) that being able to use X in this way allows us to develop applications quickly, rather than paying the starting cost of designing something from scratch. This paper is a chronicle of our efforts to date, and a summary of the lessons we have learned.

We begin with a description of the hardware, to set the stage. A fairly detailed discussion of the network layer follows, because our network environment behaves rather differently than the Ethernet for which the X protocol was designed. We follow this with a discussion of the various server implementation strategies we have tried, and some thoughts about what sorts of applications one wants to build in the ubicomp environment, and how to best use X to build them.

Hardware environment

PARC has developed devices in all three categories. The Liveboard is a 1120×780, 60"x40" back-projected display with a wireless pen interface based on Sparcstation or IBM PC hardware, very useful for meetings and distributed collaboration [Elrod92]. A product version of the Liveboard is now available from a Xerox subsidiary.

Our initial foot-sized platform was called the Scratchpad. This was a 1120×780 LCD display with a Scriptel tablet over the display, connected to a Sparcstation SBus. It allowed us to explore a device that exhibited the power of a workstation while using the tablet computer metaphor.

Our initial mobile radio platform was called the XPad. This had both a 34010 display processor and a 68302 communications processor, 4MB of ram, a 1024×768 monochrome display, and a Scriptel tablet over the display. Communication was via a 900MHz FM radio of our own design. The XPad was large, heavy, difficult to program, and there had been significant delivery problems with 1024×768 displays, even within Japan. We built three XPads, but none has become operational for more than small demonstrations.

We have now designed a new mobile radio platform we call the MPad (M for Mini). This contains a 640×480 backlit display with Scriptel tablet, a 68302 processor at a higher clockrate on an improved bus, 4MB of memory expandable to 16MB, sound in and out, and possibly other multi-media capabilities. The MPad uses a 5MHz near-field effect radio, providing cells that are approximate 6 meters in diameter. The first prototype MPad sent its first radio packet on December 20, 1991. We have built 15 MPads and deployed them in meeting areas and offices.

The ParcTab consists of a 64×128 monochrome display with pressure sensitive surface and three external buttons in a palm-fitting shape, communicating at 19,200 baud to an infrared ceiling system developed by Olivetti Research Ltd. and proposed by them as a standard for low-speed infrared communication. Inside the tab is a 8051-compatible processor, 16k EPROM, 8k NVRAM, a speaker, and an I^2C bus for peripherals. We have built approximately 50 tabs and are deploying them to members of CSL. Our experience with Tabs is described in two companion papers [Adams93a, Adams93b].

In this paper, we concentrate on our experiences with the MPad system, since the Tabs do not run X and the Liveboards act largely as workstations.

Network environment

Physical layer

MPads communicate on a wireless local area network consisting of extremely small "nanocells" about six meters in diameter. It is intended to support in-building use of hand-held wireless computing devices. It uses inexpensive, low-power, low-frequency near-field radios of our own design, which are considerably different from the 900MHz spread-spectrum technology being investigated elsewhere. The radios operate at 256 Kb/sec, with a range of 3 to 4 meters. Because they are near-field devices, power falls off very rapidly with distance, giving us very sharply-defined cell boundaries. The low operating frequency eliminates multipath effects. The relatively low data rate, is offset to some extent by the extremely small cell size, which ensures that the number of mobile stations sharing any given cell is low; in other words, the bandwidth density is of our system is comparatively high. Finally, the small cells make it possible to use routing information to provide applications with reasonably precise location sensing.

Basic Media Access Protocol (MACA)

In a wireless LAN, carrier and collision are not global properties as they are in a traditional broadcast LAN such as Ethernet. Because stations are physically separate and have limited range, the existence of carrier or a collision can be defined only with respect to an intended receiver. Thus, carrier sense and collision detection at the transmitting station don't really make sense, and we needn't be too embarrassed that our hardware doesn't provide them.

The stations work around the absence of carrier sense and collision detection by exchanging a pair of (short) control packets before sending a (long) data packet. The control packets, *RTS(n)* ("Request To Send n bytes") and *CTS(n)* ("Clear To Send n bytes") are vulnerable to collision. After a successful exchange of control packets, with high probability all relevant stations know that a data packet is about to be sent and avoid colliding with it. This strategy is also used in Apple Localtalk protocols [Sidhu90]. To our knowledge it was first suggested for wireless media access control by Karn [Karn90], where it is called MACA for "Media Access with Collision Avoidance."

Stations not directly participating in the exchange (called "bystanders" below) must refrain from sending in order not to disrupt the exchange. They do so according to the following defer rules:

D1: A bystander hearing an RTS(n) packet must defer for the time required to send the expected CTS(n) packet.
D2: A bystander hearing a CTS(n) packet must defer for the time required to send the expected n-byte DATA packet.

Intuitively, these rules are justified as follows. For each pair of stations <A, B>, we make the "reciprocity assumption" that A can hear transmissions from B iff B can hear transmissions from A; *i.e.*, there is a symmetric notion of A being in range of B. It follows that packets sent by A can collide with packets being received by B iff A is in range of B iff A can hear transmissions from B. Thus, if A hears an RTS sent by B, A should defer to avoid colliding with the expected CTS at B (rule D1). Similarly, if A hears a CTS(n) sent by B, A should defer to avoid colliding with the expected n-byte DATA packet at B (rule D2). However, if A hears an RTS sent from B to C but does not hear an answering CTS, then with high probability C either has failed to respond or is out of range of A. In either case A may legitimately send packets without disrupting communication between B and C; thus, there is no requirement that a bystander hearing an RTS(n) packet defer for the expected DATA packet.

In our environment the reciprocity assumption holds "most of the time," but it occasionally fails. For example, a nearby source of low-level noise can completely disable a station's receiver while having little effect on its transmitter. We don't yet know how important this effect will be in practice.

Stations contend for the medium by sending RTS packets until one of them successfully elicits a CTS response. Given the above defer rules, is easy to see that nearly all collisions occur between RTS packets or between an RTS and a CTS packet; a successful CTS indicates the medium has been acquired.

We use a fairly standard exponential backoff strategy. Each station maintains a backoff parameter b. A station wishing to send data chooses a number w uniformly at random in [1..*b*]. After waiting w contention slots (control packet times), if no activity has been seen the station sends an RTS and looks for a CTS in reply. If a CTS is received, the backoff parameter is reduced by an additive term β

$$b \leftarrow \text{MAX}(b - \beta, \text{MINBACKOFF})$$

and the data is sent. If no CTS is received, a collision is assumed to have occurred. The backoff parameter is increased by a multiplicative factor α

$$b \leftarrow \text{MIN}(b \times \alpha, \text{MAXBACKOFF})$$

and contention is retried.

Fairness

In practice we cannot assume every mobile station will hear every successful exchange to adjust its backoff parameter; thus, the stations' backoff parameters will not track one another precisely. This can lead to asymmetric situations in which a successful sender gets to reduce its backoff and contend more aggressively than other stations, allowing it to dominate the channel for long periods. Although this behavior keeps channel utilization high, it is particularly undesirable for our system, in which most communication involves interactive use of hand-held devices.

Our solution to this problem is to reserve a byte in the header of each MACA packet for the current backoff value. Whenever a packet is received, the receiving station adopts the backoff value from that packet. Since (by definition) every station in a cell is within range of the cell base station, and (in our current architecture) all communication involves the cell base station, it follows that the most recently successful backoff value is used with high probability by every station in the cell, avoiding the unfair behavior described above. This has been verified by simulation (up to 100 stations per cell) and experiment (5 stations per cell).

The technique of copying backoff values achieves fairness only in the sense that every station wishing to acquire the channel has an equal chance of doing so. This definition of fairness is clearly inadequate for most cellular networks, in which about half the traffic in each cell originates at a single site (the cell's base station). In our current architecture, virtually all traffic comprises a single bidirectional TCP stream per portable machine, connecting the halves of the X server. To deal with this problem, our MACA code maintains a separate output queue for each destination, and behaves as if it were runninging a separate MACA output process for each destination. If one of the simulated output processes is successful, it governs the behavior of the real station. If two simulated output processes collide, and no activity is heard before the time of the simulated collision, a JAM packet is sent. Receiving stations interpret a JAM packet as a collision, so that all stations' backoff parameters are updated consistently.

MACA acknowledgements

We were initially concerned that our radios would have a high error rate. Although this has not turned out to be as serious as was feared, it is true that the most likely place for a packet to be dropped is the wireless link [Richley93]. Moreover, the protection afforded to a DATA packet by the preceding RTS-CTS exchange is considerably reduced if there are multiple, overlapping cells. For these reasons we took the obvious step of adding acknowledgements to MACA. Now data is sent between stations A and B by a 4-way exchange:

1. Station A sends RTS(n) to station B, indicating its desire to send n bytes of data.
2. Station B replies CTS(n), indicating its willingness to accept the data.
3. Station A sends n bytes of data.
4. Station B sends ACK.

Of course the use of ACK packets requires sequence numbers. It also has a fairly subtle interaction with the defer rules. We have introduced the obvious defer rule for DATA packets:

D3: A bystander hearing a DATA packet must defer for the expected ACK packet.

This rule does not protect ACK packets to the extent that rules D1 and D2 protect CTS and DATA packets. To see this, consider three stations A, B and C, where A is within range of B and B is within range of C, but A and C are out of range. Suppose B sends an RTS to C (which is also heard by A) and B receives an answering CTS (not heard by A). While B sends its DATA, A may legitimately engage in communication with some other station; thus, a packet sent by A can collide with the ACK eventually sent by C to B. Fortunately, such a collision is not especially costly. After a link-layer timeout (on the order of a single contention slot), B resends the RTS. Since the data has already been transferred successfully, C responds with an ACK, which (like a CTS) is protected by rule D1. Thus, while some additional contention is introduced, the data is not resent, and the extra overhead is independent of the size of the data packet.

Our MACA implementation provides a callback interface, which a sending client can use to learn when a queued packet has been delivered over the radio and acknowledged (or discarded due to excessive retries). As discusseed below, this information is used in two ways: it enables our TCP header compression algorithm to maintain consistent per-connection state, and it enables our TCP implementation to coalesce short data packets in many cases.

Unsolicited Data and TCP Header Compression

Our X server sends pen events over the radio. This means that much of our radio traffic is very small data packets. This makes the overhead of a 4-way exchange particularly high, and suggests the optimistic strategy of sending (and acknowledging) unsolicited data packets if the packets are sufficiently short [Karn90]. The potential gain of sending unsolicited DATA rather than an RTS is the elimination of the RTS/CTS exchange. Unfortunately (but not surprisingly), if the DATA packet is significantly longer than a single contention slot this potential gain is completely offset by the increased collision cross section of the

unsolicited DATA packet compared to an RTS; a DATA packet containing a 40-byte TCP+IP header is too long to be sent this way. We have solved this problem by implementing a version of Van Jacobson's TCP header compression algorithm [Jacobson90] for MACA. This algorithm typically compresses TCP+IP headers to 3 or 4 bytes, resulting in a total of about 20 bytes of data in the MACA packet associated with a pen event. The measurements below show that these packets are small enough to benefit from being sent as unsolicited data.

As originally described, Jacobson's header compression requires the link layer to report receive errors to the decompressor, which then enters "toss mode" until receiving an undamaged packet containing an explicit connection number. Because collisions, which happen regularly, look to the hardware like receive errors, the MACA layer is unable to provide this information reliably. Fortunately, we can rely on the ackowledgements provided by MACA to ensure that the compressor is notified when a compressed packet may not have been delivered. It is straightforward to modifiy the algorithm so that connection state is discarded at the compressor end (rather than the decompressor end) in this case.

Coalescing TCP writes into larger packets

Even with optimizations such as TCP header compression and MACA 2-way exchange, the packet overhead for a small packet is quite large. This is a particular problem for traffic such as pen events, since they tend to be short (16 bytes) and spaced out in time (typically 50 events/second). To get adequate user response, it is important to keep the delay on delivering pen events as small as possible. However, delivering each in its own TCP packet uses up an incredible amount of channel bandwidth.

One technique for avoiding small data packets was suggested by Nagle [Nagle84]. This scheme avoids sending less than a full-sized TCP segment whenever there's already some unacknowledged data outstanding. Unfortunately, it does not work very well for a low bandwidth continuous data stream such as pen events, as it adds roughly a full round trip worth of delay to all the events and this seriously degrades responsiveness.

Instead of using Nagle's technique, our system takes advantage of the callback mechanism provided by the MACA driver. It leaves all the data queued at the TCP layer for as long as possible, creating a new packet and handing it to MACA only when the last packet has been sent on its way. Holding onto the data at the TCP layer does not introduce any extra delay, as packets are made available as fast as MACA can deliver them. In fact, since the MACA queue is kept small, it reduces the average delay. In addition, the larger packets which result improve overall throughput.

Network performance

Basic TCP throughput

Using 4-way exchanges and no header compression, and sending 512-byte data packets, we achieve a TCP throughput of 117.6 kbps. This represents a little less than half the bandwidth of our 256 kbps radio link.

To put this number in perspective, we note that our MPad MACA implementation requires approximately 4 ms for an RTS-CTS exchange, of which about 1.5 ms is accounted for by packet transmission times. The time required for a 0-byte DATA-ACK exchange plus a random backoff computation (in a lightly-occupied cell with the backoff parameter at its minimum value) is approximately 6 ms, again including 1.5 ms packet transmission times. Thus, the 4-way exchange adds 10 ms (320 byte times at 256 kbps) to the cost of sending a data packet. Using 512-byte packets we get an acknowledgement for every second TCP packet. Thus, sending 1024 bytes of data requires three 4-way exchanges (two data packets and an ACK) and sends three 40-byte TCP-IP headers. The total overhead is 960 byte times for the three 4-way exchanges, plus 120 bytes for the three TCP-IP headers, or a total of 1080 byte times of overhead to send 1024 bytes of data.

The effect of the fixed 10 ms overhead is proportionally greater as the packet size goes down: again using 4-way exchanges and no header compression, but sending 16-byte data packets (to simulate pen events), we achieve a TCP throughput of only 10.2 kbps.

Header compression

Adding TCP header compression and 2-way exchanges of unsolicited DATA packets as described above yields worthwhile performance improvement for short packets.

Using 512-byte data packets, so that TCP ack packets can be sent by 2-way exchange but TCP data packets require a full 4-way exchange, our measured TCP throughput is 121.6 kbps, an improvement of only 3.4% over the 117.6 kbps figure obtained without compression.

However, using 16-byte data packets, so that TCP data and ack packets can both be sent by 2-way exchange, our measured TCP throughput is 14.0 kbps, an improvement of 37% over the 10.2 kbps figure obtained without compression.

Fair bandwidth division

To show that MACA provides fair distribution of bandwidth among multiple stations, we ran repeated experiments in which 4 MPads sent data as fast as possible to the same base station using 512-byte TCP data packets with compression and 2-way exchange enabled. Data rates achieved by individual MPads ranged from a low of 28 kbps to a high of 37 kbps. Interestingly, there was no performance penalty as a result of sharing the channel — in all experiments the aggregate throughput was around 125 kbps, actually slightly higher than the maximum throughput achievable by a single MPad sending alone.

TCP write coalescing

To test the performance of the TCP write coalescence algorithm, we created an artifical source of "pen traffic." It generated 16 byte chunks of data on a TCP stream at a rate of 50 events/second. The resulting stream needed 800 bytes/second, or 6.4 kbps of bandwidth.

Based on the above numbers for 16 byte data packets, a stream such as this would use up roughly half of the channel if no data was coalesced. However, with coalescence, four pads in a single cell were able to simultaneously get exactly the 6.4 kbps they asked for.

Standard X traffic would involve not only pen events on the channel, but also some set of traffic in the reverse direction with drawing requests. While this reverse traffic is typically blocked into fairly large packets, the bandwidth available for it can be greatly reduced by the pen traffic. The available bandwidth for such a reverse stream without coalescence was only 53.1 kbps, or 43.7% of the bandwidth available on an otherwise idle channel. With coalescence, the available bandwidth rose to 85.6 kbps.

X servers for non-wired devices

For the purposes of discussing our X server work, there are two classes of devices - wired and non-wired. The distinction is really driven by the available bandwidth and latency of the connection of the device and the rest of the computing infrastructure.

The liveboard and scratchpad are driven by fairly straightforward ports of the MIT sample server. To the software, the liveboard looks like a monochrome frame buffer with a slightly odd input device - we have made small changes to simultaneously support the standard mouse as well as the liveboard pen. There is one liveboard that has provisions for multiple pens, and we have used the X Input Extension to support these, but have little experience with applications that can actually make use of them. Similarly, the scratchpad is just another monochrome frame buffer with a one button input device (the pen).

The various incarnations of Pads and the Tabs have been more challenging. The pad radios have all had a basic data rate of 256 kbps, of which we expected to get 120 kbps by the time we add in the overhead of media access, IP and TCP. Practical experience with X over leased 56kbps lines led us to believe that this is adequate, and our experience has shown this to be so, with the possible exception of response time for stylus input events.

Porting decisions

We wanted to get the hardware deployed quickly in order to allow developers to begin experimenting with new code. Thus, we chose a porting strategy that got applications running against the Pads quickly, but was not intended to give optimal performance.

The first such attempt was a simple bitmap difference program. This program runs on a workstation, opens the frame buffer memory, and continually reads the upper left hand section that corresponds to the size of the pad display. Each successive "frame" is bitwise compared to the previous one, and the differences are compressed and sent to the pad across the radio. Input events from a mouse attached to the pad are fed back into the server by the same program warping the workstation's cursor and using SendEvent to synthesize button presses.

There were two big problems with this approach, both having to do with the latency involved. The most obvious was that cursor tracking in this scheme was almost impossible. The latency of the path from moving the mouse to getting the feedback of the cursor having moved was almost half a second, which was most disturbing and essentially useless. The second was that the bitmap differences could get as far as 10 seconds behind, exacerbated by media contention with traffic from subsequent cursor movement and updates. Another problem was that the constant polling of the frame buffer memory was a load on the system. So the first lesson that we learned is to implement cursor tracking as close as possible to the input device, something that many hardware vendors still haven't learned, since they are using the "software cursor" code from the sample server.

Our next choice was to do a partial port of the X server. We didn't wish to port completely to the pad, for two reasons: the MIT sample server is strongly dependent on a fairly heavyweight operating system environment (in particular, it counts on having a file system to store much of its information), and we did not wish to route inter-client communication that uses the X server as a rendezvous point over the slow radio.

The sample server is made up of several "layers" in an attempt to provide the server implementer or porter great flexibility in trading off implementation effort for ultimate performance [Angebranndt88, Angebranndt90]. The lowest, device-dependent layer ("ddx") can be implemented with highly tuned code that has extremely intimate knowledge of the device. Or it can be implemented with the supplied very simple code ("mi") that counts on little about the device except how to paint or fetch a horizontal line of pixels from the display. This set of minimal routines is called the Pixblt layer.

We devised a "split server" architecture, where the Pixblt routines run on the pad, and the rest runs on a workstation, with a network connection between them. This Pixblt layer, consisting only of routines to handle the input device and draw and fetch one-bit high bitmaps, is very straightforward to implement, and the rest of the server allows a quick and dirty port by implementing just these simple routines.

We built this first with two simulations - a network throughput simulator and a pad simulator. The throughput simulator allowed us to control the available network bandwidth to simulate behavior over the radio, and the pad simulator allowed the back-end pad code to be debugged on a workstation without worrying about infant hardware environment problems. When this was all working, we moved the code to the actual hardware in short order.

Unfortunately, the throughput simulator simulated a slow, but perfect, TCP link. As detailed in the previous section, we discovered that building a reliable radio device and media access protocol is not a straightforward task. Not only is reliability of transmission a problem, the effects of small packet sizes on available channel capacity was unforeseen.

This allowed us to get some of our applications up and running, albeit slowly. The first thing that we noticed was how slow everything seemed. We had to re-learn the lesson about cursor tracking from the Xpad, and quickly implemented tracking of the cursor and changing its shape on the pad side. We then began to analyze what protocol was being sent and took steps to improve perceived performance, both by tuning our code and adding to the protocol.

Rectangles are the most important

There are three routines in the Pixblt layer: FillSpans, which fills a set of horizontal, one-bit high bitmaps (spans) according to the current fill rule (a solid color or a pattern tile); SetSpans, which fills a set of spans from a source bitmap, and GetSpans, which reads a set of spans out of the frame buffer. In order to make the code as simple as possible, FillSpans was implemented without requiring any state on the pad side. This meant that the pattern tile, if any, had to be transmitted with each request. We quickly learned what a horrible decision this was - most FillSpans requests that were user feedback (as opposed to those involved in displaying a window for the first time) involved very small spans, and time spent sending the tile dominated the time required to satisfy the painting request. By adding a small cache of pattern tiles on the pad side and a few protocol requests to manage it, we were able to paint simple 10×10 rectangles over ten times as fast.

We learned this lesson again as we added a protocol requests to fill a rectangle, rather than a series of spans, from a tile. This allowed a rectangle to be described by four 16-bit integers, instead of two 16-bit integers and a 32-bit width *per pixel of height*. This change painted 10×10 rectangles 60 times faster. Moreover, it provided a subjective shift in the "snappiness" of the system - when a window is first mapped, it is painted as a main rectangle with many smaller rectangles for borders and window manager-supplied decoration. These rectangles all were now painted much more quickly, giving the perception of much greater speed.

We learned a similar lesson when we switched the code that implemented SetSpans from sending one request per span to batching a TCP window's worth of spans before sending them. This only doubled the measured copying speed, but the visual effect was much more striking: there was a longer pause while the data was received, but then painting proceeds as fast as the CPU can execute the inner loop.

Each of these reductions in protocol overhead had a side benefit - the cursor tracking code removes the cursor from the screen just before processing a request and replaces it immediately after. Fewer protocol requests mean less time wasted repainting the cursor and even snappier response.

We currently have complaints about two areas of performance: painting text, and copying between Pixmaps (bitmaps stored in the server) and the screen. Making fonts go faster requires adding font support to the pad, which means either adding network file system access to the pad's operating system, or porting the X font-server client code to the pad. A "quick hack" might be to download fonts on the fly, but this looks like as much work as doing it right.

Pixmaps are stored on the workstation side of the split server, mostly for expediency of implementation. (Moving the pixmaps to the pad means that the server has to do management of the remote memory, and painting into pixmaps suddenly becomes slow because it has to go across the wire, or requires a lot of the drawing code to be moved to the pad, so that the painting requests can execute there.) As a result, displaying a pixmap on the screen results in a lot of SetSpans requests going over the wire, filled with bits to be painted.

We have implemented G3 compression algorithms for the SetSpans code, to try to minimize the amount of data sent. This code is currently turned off. Most of the SetSpans requests send small enough amounts of data that the effort required to decompress the bits is more than the time saved by reducing the bandwidth. The protocol overhead of sending the request is the dominant factor.

Next steps

At this point, we determined that doing window copies local to the pad, and providing direct font support (text is currently painted using mi code that breaks the glyphs into spans and sends them) are next in importance. The code and protocol changes to implement these are fairly substantial, especially since the pad has no access to a file system (to get the fonts). As the design of this next phase proceeded, we came to the realization that we were implementing more and more of the sample server on the pad, in fairly undisciplined manner.

Thus, we have decided to do a full port of the server to the pad, in order to measure the actual cost of the inter-client traffic that we worried about. We expect to be aided in this by the X Consortium's Low Bandwidth X (LBX) effort, which is attempting to address the bandwidth requirements of the core X protocol [Fulton93]. In particular, LBX intends to decrease the cost of the empty space in the X protocol encoding, and use caching techniques to reduce inter-client communication where possible.

Performance of text-based applications is quite satisfactory at this point. But our target applications, which mostly involve inking of stylus input, are still rather slow. This is because the ink is being drawn by the client program, and it takes a fairly long time for the input event to reach the client program and the feedback request to reach the display. One solution to this problem is to provide an extension that does local inking [Rhyn91, Kempf93]. We have so far resisted this option, because it means a non-standard change to the server that we have to maintain and track across releases, both in the Mpad server and our workstation servers (so applications developers can work at their desktops, one of the original motivations for using X). We prefer to exhaust tuning opportunities at all other points in the system.

Another solution is to build local clients - applications that run "native" on the pad and do not incur the overhead of radio communications before providing feedback. This is a large undertaking. Our operating system does not currently support any form of dynamic loading. Local clients must share screen real estate with remote clients: they must either be X clients, or we must provide a lower-level real estate manager that both local clients and the X server manipulate.

Sharing, migration and applications

Part of the ubicomp vision is that there are enough devices around that it no longer become necessary to carry them; wherever you go, there will be one that you can just pick up and start using, much as we treat pads of paper today. This implies that it should be easy to "whistle up" your state from display device to display device.

Doing this with X is problematic at best. When a client program first contacts a display server, it must learn a fair bit about the display hardware and tailor itself to the display. In particular, an application will determine the density of pixels on the screen, how many significant bits of color information are available, allocate some pixels out of the display's colormap, and set up some pattern tiles. These resources are identified by 32-bit identifiers unique to a connection, and are used in future protocol requests. If the client program wishes to start drawing on a different X display, it must be prepared to recreate all this state and possibly alter its operation (if, for example, there are more or fewer bits of color display available). If the client wishes to draw on two or more displays at once, it must keep track of multiple sets of resource IDs and color models. Few existing clients are prepared to do this - if the connection goes away, the client exits.

There are basically two options available to the user who wishes to migrate or share a client between several displays. The first is using a program known as a *pseudoserver*, which program sits between the application and the X server driving the display(s) [Gust88, Patterson90, Menges93]. It looks like a display server to the application, and a client to the display server. The pseudoserver may perform arbitrary transformations on the protocol requests, including remapping resource IDs and multicasting protocol requests. The second is to provide an underlying toolkit for the application which is prepared to manage multiple displays.

The primary advantage of using pseudoservers is that it works with any unmodified X application. Since the pseudoserver is dealing directly with raw X protocol it need not know anything about what the application does or how it works. There are several disadvantages however. Because applications generally connect to a server for their lifetime, if the user ever wishes to share or migrate she must connect through the pseudoserver when first starting the application. If she decides she wants to share an application that was not started with a pseudoserver, there is no way to do it without restarting. If all applications are run with a pseudoserver then the cost of the pseudoserver is paid, even if nothing is ever shared. There is also a performance cost to using a pseudoserver because it introduces an extra communication hop in the X protocol, so roundtrip requests to the server require four context switches instead of two.

X requires clients to have fairly detailed understanding of the display hardware's color model. Most clients do not make provisions for the color model to change during the connection lifetime. Thus a pseudoserver must choose carefully what color model it will provide to clients, and must be able to implement some reasonable facsimile of that color model on every display it encounters. A simple pseudoserver will force clients to run in monochrome mode, but this is not very satisfactory. An ambitious pseudoserver will offer 24-bit deep true color, and attempt to render it on whatever hardware it finds. This is very slow. Most pseudoservers offer 8-bit color and do the best they can. This is close to the approach taken by NeWS [Gosling], except that NeWS implements the mapping of the single abstraction to the hardware very close to the hardware. The pseudoserver must translate the X requests from the client in a fairly clumsy manner.

Shared window servers handle about 90% of existing applications without too much trouble. The applications that cause trouble are those that understand a lot about the display that they are running on to do more "interesting" things. An example of an "interesting" application is one that allocates planes in the colormap in order to do animation by manipulating colors; a pseudoserver can not guarantee that it will implement this correctly on displays with different color capabilities or even on different displays with the same color capabilities but different colormap contents. A different class of problem is raised when sharing or migrating among displays with widely differing resolutions. An application that starts out with a display tailored for a 72 dpi display will probably not adapt correctly to a 300 dpi display, and a pseudoserver can only do so much in this case, because distances in the X protocol are expressed in terms of pixels, not a linear measure such as points, inches or millimeters.

The second option is to write the client with a toolkit that hides the details of sharing and migration, and takes care of connection startup and shutdown, resource allocation and remapping, and so on. This toolkit will probably not expose as much detail about the hardware as the X protocol allows, and may save as much state about the client program as necessary in order to map the drawing model it provides onto the various display hardware it comes across. Dealing with these problems at the toolkit layer allows applications to look "right" on several different displays at the same time, since the application defines itself in terms of the model the toolkit provides, instead of the arbitrarily complex model available by using the full X protocol. All applications written with the toolkit gain the ability to share and migrate, and applications can be written which are aware of this and have special behavior. The applications behave in a consistent way and do not require extra effort to begin sharing. Performance is better since there is no pseudoserver involved.

We believe that most of the existing applications are not appropriate for a shared environment, or even for a pen-based environment. So a 90% solution for the simpler applications that we wish to retain is adequate. For the new applications that we will build, we are building the ability to migrate into the underlying toolkit. We have built two such toolkits: one for the Cedar environment [Jacobi92], and one which invisibly adds sharing and migration to the Modula 3 Trestle toolkit [Manasse89]. Both of these toolkits provide a (different) abstract drawing model that allows clients to express an ideal display, which is then rendered as well as possible on the available hardware.

There is a third approach to sharing state with X clients, and that is to introduce an application-specific protocol for exchanging data between clients that contact a single display. The **Tivoli** shared drawing program, developed at PARC [Pedersen93], and Van Jacobsen's **wb** program, developed at the Lawrence Berkeley Laboratories [Jacobson92] use this approach. Tivoli uses direct TCP connections between all the clients sharing a shared drawing surface (including clients that do not use X to display the surface), and wb uses multicast IP to communicate the history of a drawing session.

Conclusions

We are not at all convinced that using X for pen-based mobile/wireless computing is a good idea. The X protocol was designed to be operated on an Ethernet; it was expected that available bandwidth would be high and latency would be low and of low variance. Even on an Ethernet, application programmers must try to minimize the number of round trips from client to server. On a low speed link, this becomes even more of an issue. The X Consortium LBX effort addresses some of these problems, but it only directly deals with protocol encoding issues, not necessarily problems raised by unusual network fabrics.

Treating the pads as X terminals forces us into a computing model which makes it expensive to provide snappy input response (there is a long delay between an input event and the client program's response to that event). We can address this by running clients local to the pad, but this means that the pad must support a fairly heavyweight computing environment, complete with access to network and file resources. We can address it by building an extension to the server that does local inking or local input response, but this means that we must support a non-standard body of code across changes to the X server. None of these solutions provides us with a way to give programs running entirely on the pad direct access to the pad's display; another modification to the X server is needed.

We will continue down this path for a while longer, because the plus side is that the development environment is familiar to us, and allows us to deploy our applications quickly across all our platforms. Developing

a new window system and deploying it is an expensive proposition, and one which we are not willing to enter into lightly.

Acknowledgements

Lawrence Butcher developed the networking basestations. Ed Richley designed and built our radios, and Melvin Chan keeps them running. Steve Deering made mobile IP work. Tom Rodriguez (a summer intern from Georgia Tech) resurrected JoinVBT in the distributed Trestle toolkit and made it work in X across hardware with different color models.

References

[Adams93a] Norman Adams, R. Gold, B.N. Schilit, M. Tso and R. Want. 'An infrared network for mobile computers'. *Proceedings USENIX Symposium on Mobile and Location-independent Computing, Cambridge Massachussetts*. August 1993.

[Adams93b] Norman Adams, R. Gold, B.N. Schilit, M. Tso and R. Want. The **PARCTAB** mobile computing system. Xerox Palo Alto Research Center. In preparation.

[Angebranndt88] Susan Angebranndt, R. Drewry, T. Newman and P. Karlton. 'Strategies for porting the X11 sample server'. Software Distribution Center, Massachusetts Institute of Technology, Cambridge, MA. 1988

[Angebranndt90] Susan Angebranndt, P. Karlton, R. Drewry and T. Newman. 'Writing tailorable software', *Software-Practice and Experience*, **20**:S2/109-S2/118, 1990.

[Elrod92] Scott Elrod, R. Bruce, R. Gold *et al.* Liveboard: a large interactive display supporting group meetings, presentations and remote collaboration. Xerox Palo Alto Research Center. Report CSL-92-6, 1992.

[Fulton93] Jim Fulton and C. Kantarjiev. 'An update on low bandwidth X (LBX); a standard for X and serial lines'. *Proceedings of the 7th annual X Technical Conference, January 1993*. O'Reilly & Associates, 1993.

[Gosling89] James Gosling, D. Rosenthal and M. Arden. *The NeWS Book: an introduction to the networked extensible window system.* Springer-Verlag, 1989.

[Gust88] Phil Gust. 'Shared X: X in a Distributed Group Work Environment'. Unpublished paper presented at the second annual X Technical Conference, January 1988.

[Jacobi92] Christian Jacobi. 'Migrating widgets'. *Proceedings of the 6th annual X Technical Conference, January 1992*. O'Reilly & Associates, 1992.

[Jacobson90] Van Jacobson. 'Compressing TCP/IP headers for low-speed serial links'. Arpanet Working Group Requests for Comment, DDN Network Information Center, RFC-1144. February 1990.

[Jacobson92] Van Jacobson. A portable, public domain, network whiteboard. Presentation at the Xerox Palo Alto Research Center, April 28, 1992.

[Karn90] Phil Karn. 'MACA - a new channel access method for packet radio'. *Proceedings of the ARRL 9th Computer Networking Conference, London Ontario, Canada.* September 22, 1990.

[Kempf93] James Kempf and A. Wilson. 'Supporting mobile, pen-based computing with X; mobile information for hi-tech nomads'. *Proceedings of the 6th annual X Technical Conference, January, 1992.* O'Reilly & Associates, 1992.

[Manasse91] Mark S. Manasse and G. Nelson. Trestle reference manual. Digital Systems Research Center. Research Report 68, 1991.

[Menges93] John Menges. 'The X engine library: a C++ library for constructing X pseudo-servers'. *Proceedings of the 7th annual X Technical Conference, January 1993.* O'Reilly & Associates, 1993.

[Nagle84] John Nagle, 'Congestion control in IP/TCP internetworks'. Arpanet Working Group Requests for Comment, DDN Network Information Center, RFC-896. January 1984.

[Patterson90] John F. Patterson. 'The good, the bad and the ugly of window sharing in X'. Unpublished paper presented at the fourth annual X Technical Conference, January 1990.

[Pedersen93] Elin Rønby Pedersen, K. McCall, T. Moran and F. Halasz. 'Tivoli: An electronic whiteboard for informal workgroup meetings'. *Human Factors in Computing Systems: Proceedings of InterCHI 1993.* Association for Computing Machinery, 1993.

[Rhyn91] Jim Rhyne, D. Chow, M. Sacks. 'The portable electronic notebook'. Unpublished paper presented at the fifth annual X Technical Conference, Cambridge, MA. January 16, 1991.

[Richley93] Ed Richley and S. Elrod. Experiments with near-field radios. Xerox Palo Alto Research Center. In preparation.

[Scheifler92] Robert W. Scheifler and J. Gettys. *X Window System.* Digital Press, Bedford, MA, 1992.

[Sidhu90] Gursharan S. Sidhu, R. Andres, and A. Oppenheimer. *Inside AppleTalk®.* Addison-Wesley, 1992.

[Weiser91] Mark Weiser. 'The computer for the 21st century'. *Scientific American* **265**(3):93-104, September 1991.

Author Information

Christopher Kent Kantarjiev is a Member of the Research Staff in the Computer Science Lab at the Xerox Palo Alto Research Center (in Palo Alto, CA, natch). He has a B.S. in Physics from Xavier University (1979) and a Ph.D. in Computer Sciences from Purdue University (1986). He is a frequent contributor to the magazine of the Vintage Triumph Register. He can be reached as cak@parc.Xerox.COM.

Alan Demers is a Principal Scientist in the Computer Science Lab at the Xerox Palo Alto Research Center. He has a B.S. in Physics from Boston College (1970) and a Ph.D. in Computer Science from Princeton University (1975). He can be reached as demers@parc.Xerox.COM.

Ron Frederick is a Member of the Research Staff in the Computer Science Lab at the Xerox Palo Alto Research Center. He has a B.S. in Computer and Systems Engineering from Rensselaer Polytechnic Institute (1989) and an M.S. in Computer Science from Stanford University (1991). He can be reached as frederick@parc.Xerox.COM.

Robert T. Krivacic is a Member of the Research Staff in the Computer Science Lab at the Xerox Palo Alto Research Center. He has a B.S. in Computer Science from Bowling Green State University (1974) and a M.S. in Computer Science from The University of Colorado (1978). He can be reached as krivacic@parc.Xerox.COM.

Mark Weiser is the head of the Computer Science Lab at the Xerox Palo Alto Research Center. He has a PhD in Computer and Communication Sciences from the University of Michigan (1979). His hobbies are existential philosophy, sociology of science, and writing computer games; his recent technical work has been on garbage collection and ubiquitous computing. He can be reached as weiser@parc.Xerox.COM.

Customizing Mobile Applications

Bill N. Schilit *
Computer Science Department
Columbia University
schilit@parc.xerox.com

Marvin M. Theimer
Palo Alto Research Center
Xerox Corporation
theimer@parc.xerox.com

Brent B. Welch
Palo Alto Research Center
Xerox Corporation
welch@parc.xerox.com

ABSTRACT

The dynamics of mobile systems require applications to intelligently adapt to changes in system configurations and to their environment. We describe a workplace in which users interact with a number of stationary and mobile systems through the course of a day. The relationship between systems and devices is constantly changing due to user mobility. We present a facility for mobile application customization called "dynamic environment variables." The facility allows flexible sharing of customization contexts, supports short interactions with long term services, and provides efficient notification of environment changes to applications. A sample application built in PARC's mobile computing environment and initial performance evaluations are described.

1 Introduction

Mobile computing differs from desktop computing because of the dynamic nature of system state. In our research lab mobile users interact with mobile computers as well as a world of stationary and embedded systems [8]. As users move, the sets of stationary and mobile objects they deal with changes. This dynamic state is not just a problem, it is also an opportunity for applications to intelligently adapt to their environment.

This paper describes a facility for *dynamic customization* of applications. In contrast to the Unix practice of one time initialization at program start up, dynamic customization supports changing application preferences at any time during program execution. In particular, the system modifies application preferences in response to changes in the physical computing environment.

Most software is designed with the assumption that the system it operates in will not change much over time. Computers are configured according to the hardware on the desktop, the printers down the hall, and the file servers on the local network. Each computer is tightly knit into a particular environment. It is therefore not surprising that Unix programs, for the most part, have configurations that are difficult to change. Frequently, the only time that a configuration can be set is at program startup.

In contrast, here are some examples where changing an application's customization values after startup is necessary:

- Selection of proximate devices such as printers and displays.

*Currently visiting Xerox Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA 94304

- Application response to migrating X windows among different kinds (i.e. color and monochrome) of displays.
- Application response to being "docked" at high speed network taps versus operating over slow speed links.
- Managing and moderating resources, such as local disk space depending on network performance.
- Supporting security in an application by means of auto-lockout based on the user's location relative to his mobile computing devices.

We believe that existing mechanisms for program customization limit the usefulness of mobile applications. Consequently in this paper we introduce a general design for dynamic reconfiguration of applications. The next section summarizes Unix techniques for initialization and customization and explains why these are not adequate for many mobile computing applications. Section three presents our approach based on a community of environment variable servers. This is followed by a more detailed description of interfaces, commands, server interaction and performance.

2 Unix Initialization and Customization

Application *customization* is the setting of parameters (*preferences*) by users in order to control the appearance and function of an application. Often, user preferences are expressed at *initialization time* when the application starts running. Unix programs employ a few different facilities for program customization:

- command line arguments
- environment variables
- "rc" initialization files
- databases like Xrdb [1]

Customization by environment variable dates to the era of "dumb terminals" before window based workstations. In this situation a user has one screen and runs one process at a time. The inheritance of environments from parent process to child was sufficient because all work was carried out in a subprocess of the interactive shell. This mechanism has survived into the "windowing workstation era" but its limitations are evident to anyone who has tried to maintain a uniform environment among multiple shell windows. To some extent a workstation wide database, like Xrdb, acts as a replacement for environment variables in a windowing system.

We are entering a third era of computing in which users interact with a multitude of stationary and mobile systems through the course of a day. Our lab has built electronic whiteboards, mobile stylus-based notebooks, and personal communication devices that are in a constantly changing relationship to each other because of user mobility. Mobile systems need to maintain and share a context that is not limited to the boundary of process hierarchy, window manager, or host.

In addition, applications must separate customization from initialization if they intend to support environment changes during program execution. Historically most applications have run from start to stop in a short time frame. Today, however, we run applications, such as mail and emacs, that persist over very long time periods. We evolve a context of windows, editor buffers, command history, et cetera, which is expensive to recreate. As a consequence, mobile computing requires quick instantiation of a context as well as persistence across sessions. For example, a context that follows the user is useful for *migrating application windows*, where windows move from

[1] Xrdb, the X resource manager, is the part of X11 window system that manages user preferences about colors, fonts, etc. Each X11 screen has an associated Xrdb server process.

one display to another following a mobile user. For security reasons it is desirable that a user walking away from a device will cause the session on that device to become suspended, logged-out or locked out.

Unix applications provide an approach to separating initialization from configuration. Non-interactive servers, by convention, re-read configuration files when they receive the **HUP** signal; X window managers (like twm) support the same facility through a menu command. In both cases, reconfiguration requires manual intervention and is often heavy-handed, with the entire internal state being destroyed and recreated. There are two problems with this approach. First there is no facility for incremental changes and therefore reinitialization resets all state where perhaps only some should change, and second, it is based on configuration files that need to be changed, and hence constantly monitored by interested parties.

The X windowing system provides a server-based resource manager, Xrdb [3], for communicating user preferences to applications. The advantage of Xrdb is that clients accessing a central server do not need a common or duplicate collection of initialization files on all hosts. Although the database-server approach has the advantage that it allows changing parameters without editing files on disk, it does not notify existing applications of those changes. One could imagine extending the notion of X-events or Xrdb to solve these problems. A new type of X-event, for example, could indicate what environment information has changed. However, this solution implies that applications would require X servers on any hosts on which they are run.

Rather than focus on a limited solution, we describe a general RPC-based one in the following section.

3 A Community of Dynamic Environment Servers

A *dynamic environment server* is a program that manages a set of variable names and values (an *environment*) and delivers updates to clients that have previously shown interest by *subscribing* to the server. Typically there is one environment for each user, plus environments for rooms, work groups, et cetera. Clients use RPC to get variables from servers, and servers use RPC to provide callbacks to clients. The key feature of an environment server is that environment changes are quickly seen by multiple processes through the use of RPC callbacks. The environment servers maintain a shared context for mobile applications that is not limited to a process hierarchy or a particular window server. More important, however, is that environment servers can help model the dynamics of a mobile computing environment by providing a simple and standard interface for change notification.

We've designed an architecture that includes a community of environment servers used in conjunction with a *user agent* [7] managing user information. Environment variables describing a user are maintained by the user agent, which also keeps track of the user's current location. Applications subscribing to the user's environment will automatically be notified of changes, including their owner's location.

As an example, consider two environment variables **LOCATION** and **NEAREST_PRINTER** and how they might be used. As a mobile user moves, his location is tracked by the user agent, which updates **LOCATION** in the user's environment. Applications subscribing to this environment get notified of, among other things, location changes. Applications generally subscribe to the user's environment as well as a number of environments based on the user's current location, for example, a room. As the room changes, an application cancels subscriptions from the previous room and subscribes to new environment servers for the current room. An environment variable in such a server might be **NEAREST_PRINTER**.

It is important to remember that environments are dynamic and their contents may be frequently changing. For example, the environment for a room might include a variable for the occupants:

```
OCCUPANTS=adams:schilit:theimer:weiser:welch
```

This variable would be maintained by updates from user agents since they are the source of location information about a particular user. As the `OCCUPANTS` variable changes in the room's environment server, programs subscribing to the room are notified and may perform any re-customization that is applicable.

In summary, we are building a system of environment servers that has the following structure:

- Multiple environment servers run on the network. There is one environment for each mobile user and other environments are maintained for locations (e.g. rooms and offices).
- For efficiency there may be more than one environment per environment server.
- The location of the user is stored in the user's environment variable `LOCATION` maintained by the user agent.
- When the `LOCATION` variable changes applications are notified and they may choose to subscribe and/or drop existing subscriptions to environments based on the changed user environment .
- Subscribing to new environments is facilitated by attributed-based names. Environments that depend on a location are registered with attributes for the specific location.

4 Dynamic Environment Service

The generic interface presented here is similar to Unix environment variables that have a key and a value. The interface differs from Xrdb and Unix environment variables in two ways: (1) distribution: sharing is possible between process, jobs, and machines (2) asynchronous notification of change.

4.1 RPC Interface

The dynamic environment interface is an RPC based service consisting of the following functions:

- **`subscribe(e,cl)`**

  Request notification, via a callback to closure `cl`, when changes occur in the specified environment `e`. The closure specifies the procedure to be called and its parameters.

- **`cancel(e)`**

  Cancel a subscription and stop receiving notifications about changes to the environment.

- **`get(e)`**

  Fetch the entire environment `e`.

- **`setenv(e,k,v)`, `getenv(e,k)`, `unsetenv(e,k)`**

  Similar to the shell functions for environment variables.

4.2 Attribute-based Naming

This section describes how connections are established between clients and environment servers. A name server, described below, is used by clients to locate an environment server. Clients then issue a subscribe RPC and start receiving all values from the server. In our system configuration we expect a small number of variables per server, however, for scaleability it might be necessary to add an extra step to the subscription process whereby clients select the subset of variables that are desired.

The system design of mobile clients and many servers requires a flexible way to bind one to the other. Our approach is for environment servers to be found using associative matching on

attributes. For this purpose we use an attribute based naming system. This works as follows. Dynamic environment servers are assigned a set of attributes. If items maintained by the server can be thought of as having a location, then one attribute for the server is "location," whose values are a canononical form used throughout our lab. Subscribing to servers is accomplished by matching a subset of these attributes.

Attribute based naming has a number of advantages over other styles of naming for mobile computing. First, naming by a client's requirements rather than a server's host name makes for more "portable" systems. This allows flexible binding to servers by attributes instead of by host names or other physical characteristic that may change over time. Also, attributes allow matching by as few or many properties as is best suited to the application at hand.

The lookup specification and the registration for an item have the same form: an expression of keys, values, and nested keys and values. The specification, used for looking up an item, may contain fewer key-value pairs than the registration. In this case only key-values present in the specification determine the correctness of the match. More formally, the expression syntax is:

```
EXPR    ::= value | ( LIST )
LIST    ::= LIST PAIR
PAIR    ::= ( key EXPR )
```

where **key** and **value** are text strings. For example:

```
((Type environment) (Kind printing) (Location ((ID "35-2172"))))
```

can be used as a query to find a server exporting a set of "printing" environment variables for a room with ID 35-2172. These might include:

```
PRINTER=Snoball
NEARBY_PRINTERS=Snoball:BerkeleyBarb
```

4.3 Shell Commands

This section describes a set of Unix commands for maintaining dynamic environment variables. These commands are similar to and modeled on the shell's environment variable commands. The differences are that dynamic environment commands can specify the server (defaults to the user's server), and the command to print a dynamic environment has an option like `tail(1)` for following the changes to a dynamic environment.

- **setdenv [-env EXPR] [VAR [= word]]**

  With no arguments, **setdenv** displays all dynamic environment variables from the environment server registered as **$USER**. With the **VAR** argument, it sets the environment variable **VAR** to have an empty (null) value. With both **VAR** and **word** arguments, **setdenv** sets the environment variable **VAR** to the value word, which must be either a single word or a quoted string.

  After setting the dynamic environment variable all processes subscribing to the environment are notified. [2]

  If the **-env** switch is specified then an environment server matching the attributes specified by **EXPR** (as described in section 4.2) is used. For example:

  ```
  setdenv -env '((User brown) (Location Library))' biff = n
  setdenv -env '((User brown) (Location Library))' menu = lookbib
  ```

[2] Notification is asynchronous so completing the **setdenv** command does not imply that clients have received notification.

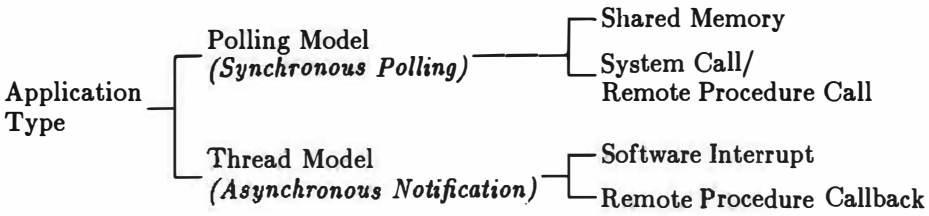


Figure 1: Dynamic Environment Interaction

will set the variable **biff** and set the variable **menu** in all environments that have the **(User brown)** and **(Location Library)** attributes. These commands might be used to disable mail notices and to add the **lookbib** program to a menu set while user brown is in the library.

- **unsetdenv [-env EXPR] VAR**

  Removes a variable from the environment. After unsetting the dynamic environment variable all processes importing the environment are notified.

  If the **-env** switch is specified then an environment server matching the attributes specified by **EXPR** is used.

- **printdenv [-f] [-env EXPR] VAR**

  printdenv prints out the values of the variables in the environment. If a variable is specified, only its value is printed.

  If the **-f** switch is specified then the program enters an endless loop, reporting any changes to environment variables as they occur.

  If the **-env** switch is specified then an environment server matching the attributes specified by **EXPR** is used.

4.4 Dynamic Application Design

In this section we describe the types of modifications that are necessary for applications to support our approach. The way dynamic environment variables are introduced into applications depends on the existing application framework. Environment changes occur asynchronously to the executing application; however, applications unprepared for concurrency (e.g. lacking locks) cannot easily field asynchronous notifications. To address this we provide different interfaces for applications designed with concurrency in mind and those that are single threaded.

Figure 1. categorizes the software mechanisms available to Unix applications for inter-process communication. These reflect the design choices for supporting a Unix based dynamic environment that relies on communicating values over the network or between processes. The figure is divided into techniques for multi-threaded applications capable of concurrency, and applications built with a polling model and not capable of concurrency. Single threaded clients can access the dynamic environment by polling for changed values from shared memory, by invoking a system call (e.g. to read a file) or by issuing a remote procedure call. Concurrency-capable applications can be notified asynchronously by software interrupt, remote procedure callback, or similarly from input available on a non-blocking file descriptor.

Asynchronous notification has the advantage that the application is immediately informed of changed values and no cycles are wasted on polling. However, for this to work, applications must be multi-threaded. For concurrent applications our system provides an RPC callback interface. Non-concurrent applications use a polling interface based on a shared memory server. These approaches are described below.

Concurrent applications access the dynamic environment through an RPC connection and an RPC callback connection. A small library hides the RPC layer from applications. We support both C/C++ and Modula-3 programming with separate versions of this library. One issue addressed by the library is how to integrate a potentially blocking RPC callback service with an applications. The Modula-3 language [6] has built-in thread support that we take advantage of by dedicating a thread to the callback RPC service. Since there is no standard threads definition for C/C++ a main loop containing the `select` or `poll` system calls is often used to provide a poor man's multi-tasking. Our C/C++ library supports this approach by allowing the main loop in either the library or in the application.

Single threaded applications access the dynamic environment through shared memory [5, 4]. An application maps a memory segment containing the dynamic environment into its address space and periodically polls the contents for changes. A separate process is used to receive dynamic environment updates from the network and write them into the shared memory. The cost of this extra process can be amortized over multiple clients. Furthermore, polling shared memory can be much more efficient than polling techniques requiring a system or remote procedure call. As in the concurrent model, a small library hides the shared memory layer from the application. The non-threaded library supports C/C++ programming.

4.5 Example Application

One of the applications we have built uses dynamic environment variables for presenting a proximate device selection dialog to the user and then migrating the user's editor window upon selection. This application encorporates two uses of the dynamic environment. The dialog employs the dynamic environment to update a presentation list of nearby displays, and separately, the editor window changes displays whenever the the `DISPLAY` dynamic variable changes value. This process is described more fully below.

A ParcTab [2, 1] based application, the *proximate chooser* presents a selection of nearby displays for migration. The ParcTab is a hand held wireless device that communicates with applications running on back-end hosts.

The chooser initially shows a proximity ordered list of displays from the `NEARBY_DISPLAYS` dynamic environment variable. This variable is part of a room's dynamic environment. The application library monitors the user's location and withdraws subscriptions that were based on the old location and subscribe to the appropriate servers based on the new location. This process causes the `NEARBY_DISPLAY` variable to change as the user moves from room to room. The chooser, a Modula-3 application, uses an RPC callback thread to obtain environment notifications. After the dialog starts, movement between offices or common areas causes updates to the `NEARBY_DISPLAYS` variable. Changes to this variable in turn update the chooser dialog with a new proximate selection ordering. When the user decides they want to setup shop at a display they make a selection which the chooser uses to set the `DISPLAY` dynamic environment variable.

Changing the `DISPLAY` variable has the effect of propogating the new value to applications that are interested in display migration. We use an extended Emacs editor supporting a command for moving the editor's X window to another display. Our Emacs interface to dynamic environment variables uses a subprocess to run `printdenv` in the mode where it enters a loop reporting variables changes as they occur. The variables are read by Emacs from a pipe and converted into calls to user supplied elisp functions. The use of filter procedures invoked on process output is a standard Emacs technique. The mechanism is based on polling a file descriptor for available input that is classified in a polling model. One filter function for processing dynamic environment variables watches for changes to the `DISPLAY` variable and causes the editor window to migrate to the new display.

4.6 Performance

The architecture of our system encourages *cliques* of clients and servers within a larger client-server community. A clique of clients, for example a set of applications in the same room, will

subscribe to change notifications from a single server. To determine the performance of the system we have measured a single clique. For the shared memory server, we believe the system load is determined by the same costs as the callback server. This is because the additional expense of a local procedure call to poll variable changes stored in shared memory is small compared to the cost of system and remote procedure calls as long as the polling rate is suitably low.

The most important parameters that determine an environment server's performance are the number of client subscriptions and the frequency of variable changes. For a timing experiment, we setup a system configuration consisting of a remote environment server, a local process setting environment variables at a Poisson distribution around a fixed rate, and one or more local subscribing processes. Both hosts are Sun SPARCStation 2 class machines. We measured the variable propogation time between a set and the resulting subscribe notification. The information flow for this measurement includes a remote RPC to the environment server and another remote RPC from the environment server to each subscriber.

Figure 2. shows average notification times for an environment server supporting a collection of between 1 and 64 subscribers. The average variable update rate was between 1 and 4 updates per second with one variable being stored in the server. Similar runs with 1000 and 5000 variables at the server showed no significant timing difference indicating that RPC processing overshadows table lookups at the server. The delay for a single subscriber receiving notifications of a variable update is around 10 ms. The slope of the line shows a constant factor of around 3 ms for each additional subscriber.

There are two limits of the server, both are a result of the decision to use Sun RPC instead of a special purpose protocol. First, the number of subscriptions cannot exceed the process file descriptor limit, currently 256. Second, since unicast is used for outgoing messages, the product of update rate times subscribers is limited. Measurements show the server process capable of delivering around 80 messages/second.

Our system requires an initial associative name lookup before a subscription is made. The time for creating a subscription (including the associative name lookup of the server) is around 350 milliseconds and about half that, 150 milliseconds, if the server being subscribed is on the local host.

5 Conclusions

We have presented a simple and generic approach to handling the dynamics of a mobile computing environment. The system consists of a community of servers that each provide a set of variables and values as well as notification of changes to the set. Servers can be organized to represent physical environments, such as rooms, or work contexts such as meetings and project groups with equal ease. A source of location information feeds into the system permitting applications to recustomize according to changes in the physical environment.

The system structure we support is flexible. Unlike other approaches for sharing environment information we do not limit cooperation. Traditional environment variables limit cooperation based on process hierarchy, and Xrdb limits cooperation based on window system. These criteria restrict the class of application permitted in a mobile system that can easily span multiple machines. Alternatives such as extending X window system events might be suitable for some classes of applications. However, this requires that applications using the system must all include the X window system. In addition, this alternative does not include a way to link information about the changing physical world (such as people's locations) to changes in the values used for application customization.

Our system is scalable. Using logically distinct servers that each manage a small part of the environment means that the system can grow gradually. Moreover, the technique of binding and unbinding to environment servers permits transparent movement between administrative domains. Whether the servers are implemented as distinct processes is not exposed thereby making the potential benefit of consolidation possible. The performance measurements shows that large variable sets with reasonably high update rates can be supported. The least scalable aspect of the implementation

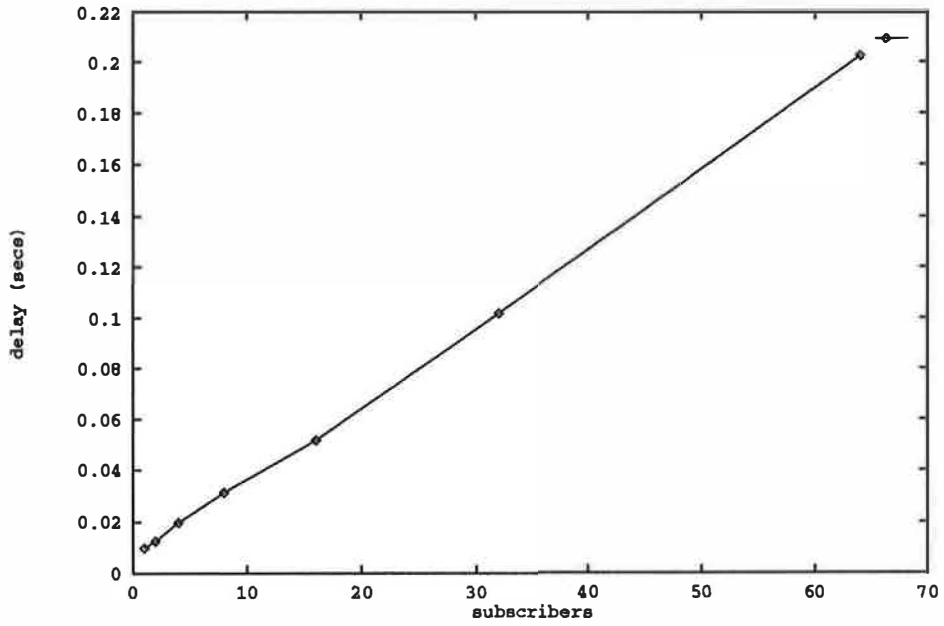


Figure 2: Variable propogation delays

is the additional cost per client-subscription at an environment server. Because our implementation is based on Sun RPC we incur the overhead of an RPC connection and the cost of an RPC message. A larger system might explore lighter weight alternatives. However, in terms of human perception the overhead is acceptable: an environment server maintaining 60 callbacks subscriptions with a variable update rate between 1 and 4 times per second has a little over a fifth of a second variable propogation delay.

Finally, our facility supports short interactions with long term services. When starting programs it is often necessary to setup a context. This may involve reading and processing shell login and other initialization files. In a mobile setting, interactions may be very short lived as a user moves between one system and another. Dynamic environments help alleviate the startup costs by preserving program context across machines and logins.

References

[1] N. Adams, R. Gold, B.N. Schilit, M. Tso, and R. Want. An infrared network for mobile computers. In *Proceedings USENIX Symposium on Mobile & Location-independent Computing*. USENIX, August 1993. To Appear.

[2] N. Adams, R. Gold, B.N. Schilit, M. Tso, and R. Want. The PARCTAB mobile computing system. Publication in preparation, 1993.

[3] Paul J. Asente and Ralph R. Swick. *X Window system toolkit : the complete programmer's guide and specification*. Digital Press X-motif series. Digital Press, Rockport, MA, 1990.

[4] M. Young et al. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proc. Eleventh ACM Symp. on Operating System Principles*, pages 63–76, November 1987.

[5] R. A. Gingell, J. P. Moran, and W. A. Shannon. Virtual memory architecture in sunos. In *Proc. Summer 1987 USENIX Technical Conf.*, pages 81–94. USENIX, June 1987.

[6] Greg Nelson, editor. *System Programming with Modula-3*. Prentice Hall Series in Innovative Technology. Prentice Hall, 1991.

[7] Mike Spreitzer and Marvin Theimer. Architectural considerations for scalable, secure, mobile computing with location information. Publication in preparation.

[8] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, September 1991.

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

* sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
* fostering innovation and communicating both research and technological developments,
* providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX is well known for its twice-a-year technical conferences, accompanied by tutorial programs and vendor displays. Also sponsored are frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter *;login:*, the refereed technical quarterly, *Computing Systems*, and has expanded its publishing role in cooperation with The MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings. News of standards efforts and reports of many meetings are reported in *;login:*.

SAGE, the System Administrators Guild

The System Administrators Guild (SAGE) is a Special Technical Group within the USENIX Association devoted to the furtherance of the profession of system administration. SAGE brings together system administrators for professional development, for the sharing of problems and solutions, and to provide a common voice to users, management, and vendors on topics of system administration.

A number of working groups within SAGE are focusing on special topics such as conferences, local organizations, professional and technical standards, policies, system and network security, publications, and education. USENIX and SAGE will work jointly to publish technical information and sponsor conferences, tutorials, and local groups in the systems administration field.

To become a SAGE member you must be a member of USENIX as well.

There are six classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided. A description of these classes is included in this packet.

USENIX Association membership services include:

* Subscription to *;login:*, a bi-monthly newsletter;
* Subscription to *Computing Systems*, a refereed technical quarterly;
* Discounts on various UNIX and technical publications available for purchase;
* Discounts on registration fees to twice-a-year technical conferences and tutorial programs and to the periodic single-topic symposia;
* The right to vote on matters affecting the Association, its bylaws, election of its directors and officers;
* The right to join Special Technical Groups such as SAGE.

For further information about membership, conferences or publications, contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

Email: *office@usenix.org*
Phone: +1-510-528-8649
Fax: +1-510-548-5738

ISBN 1-880446-51-0